SEARCH-BASED STRESS TEST: AN APPROACH APPLING EVOLUTIONARY ALGORITHMS AND TRAJECTORY METHODS

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
OF UNIVERSIDADE DE FORTALEZA
(UNIFOR)
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF SCIENCE

Francisco Nauber Bernardo Gois
June 2017

I certify that I have read this thesis and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Science.

D.Sc. Pedro Porfírio Muniz de Farias (UNIFOR) Principal Adviser

I certify that I have read this thesis and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Science.

D.Sc. André Luís Vasconcelos Coelho (UNIFOR) Co-Adviser

I certify that I have read this thesis and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Science.

D. Sc. João Paulo Pordeus Gomes (UFC)

I certify that I have read this thesis and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Science.

D. Sc. Pedro de Alcantara dos Santos Neto (UFPI)

I certify that I have read this thesis and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Science.

Ph.D. Americo Tadeu Falcone Sampaio (UNIFOR)

I certify that I have read this thesis and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Science.

Docteur Napoleão Vieira Nepomuceno (UNIFOR)

Preface

This thesis tells you all you need to know about...

Acknowledgments

I would like to thank...

Contents

List of Tables

List of Figures

Related Publications

The following publications are related to this thesis:

- **N. Gois, P. Porfirio and A. Coelho**. A multi-objective metaheuristic approach to search-based stress testing. In Proceedings of the 2017
- **N. Gois, P. Porfirio, A. Coelho, and T. Barbosa**. Improving Stress Search Based Testing using a Hybrid Metaheuristic Approach. In Proceedings of the 2016 Latin American Computing Conference (CLEI), pages 718–728, 2016 [43].

Chapter 1

Introduction

This chapter briefly introduces this work. It presents the motivations, the objectives and the structure of this work.

Performance problems such as high response times in software applications have a significant effect on the customers' satisfaction. The explosive growth of the Internet has contributed to the increased need of applications that perform at an appropriate speed. Performance problems are often detected late in the application life cycle, and the later they are discovered, the greater the cost is to fix them. The use of stress testing is an increasingly common practice owing to the increasing number of users. In this scenario, the inadequate treatment of a workload, generated by concurrent or simultaneous access due to several users, can result in highly critical failures and negatively affect the customers' perception of the company [55] [65] [100].

Software testing is an expensive and difficult activity. The exponential growth in the complexity of software makes the cost of testing to only continue to rise. Test case generation can be seen as a search problem. 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. Search-based software testing is the application of metaheuristic search techniques to generate software tests cases or perform test executions [2] [40].

Search-based testing is seen as a promising approach to verify timing constraints [2]. A common objective of a load search-based test is to find scenarios that produce execution times that violate the specified timing constraints [86]. 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, for example, must devote substantial resources to setting up and the environmental conditions are likely to be substantially different.

Usually, search-based test methods are based on single objective optimization. Multi-objective evolutionary algorithms (MOEAs) are widely used for solving multi-objective problems (MOPs) because they produce a complete set of solutions in a single run. The NSGA-II is a genetic algorithm (GA) based on obtaining a

new offspring population from the original one by applying the typical genetic operators (selection, crossover, and mutation); then, the individuals in the two populations are sorted according to their rank, and the best solutions are chosen to create a new population.

This paper addresses the validity of using a multi-objective algorithm in stress testing problems. A tool named IAdapter (www.iadapter.org, github.com/naubergois/newiadapter), a JMeter plugin for performing search-based load tests, was extended [43]. One experiment was conducted to validate the proposed approach. The experiment uses the NSGA-II algorithm with one objective: discover application scenarios where there is a high response time for a small number of users. The relevance of finding scenarios with high response times is to enable corrective actions before the application under test is released in a production environment.

1.1 Motivation

There is strong empirical evidence, that deficient testing of both functional and nonfunctional properties is one of the major sources of software and system errors. In 2002, NIST report found that more than one-third of these costs of software failure could be eliminated by an improved testing infrastructure. Automation of testing is a crucial concern. Through automation, large-scale thorough testing can become practical and scalable. However, the automated generation of test cases presents challenges. The general problem involves finding a (partial) solution to the path sensitization problem. That is, the problem of finding an input to drive the software down a chosen path [49] [26].

Software performance is a pervasive quality, because it is affected by every aspect of the design, code, and execution environment. Performance failures occur when a software product is unable to meet its overall objectives due to inadequate performance. Such failures negatively impact the projects by increasing costs, decreasing revenue or both [93].

Software testing is a expensive and difficult activity. The exponential growth in the complexity of software makes the cost of testing has only continued to rise. Test case generation can be seen as a search problem. 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. Search-based software testing is the application of metaheuristic search techniques to generate software tests cases or perform test execution [2] [40].

Performance testing of enterprise applications is manual,laborious, costly, and not particulary effective. When running many different test cases and observing application's behavior, testers intuitively sense that there are certain properties of test cases that are likely to reveal performance bugs. Distilling these properties automatically into rules that describe how these properties affect performance of the application is a subgoal of our approach [45].

Experimentation is important to realistically and accurately test and evaluate search based tests. Experimentation on algorithms is usually made by simulation. 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, for example, must devote substantial resources to setting up and the environmental conditions are likely to be substantially different.

Below we briefly describe the related research and practices on stress testing:

1.1.1 State of Research on the Search-Based Stress Testing

In the academic context, a number of studies proving the efficacy of metaheuristics to automate test execution can be found in literature. 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. However, most research studies are limited to making prototypes [2].

Garousi and Alesio presents functional tools to Systems from safety-critical domains. Garousi presented a stress test methodology aimed at increasing chances of discovering faults related to distributed traffic in distributed systems. Alesio describe stress test case generation as a search problem over the space of task arrival times. The research search for worst case scenarios maximizing deadline misses where each scenario characterizes a test case. The paper combine two strategies, GA and Constraint Programming (CP) [37] [39] [27] [28] [4].

1.1.2 State of Industrial Practices on Stress Tests

The stress testing process in the industry still follows a non-automated and ad-hoc model where the designer or tester is responsible for running the tests analyzing the results and deciding which new tests should be performed [58].

Typically, performance testing is accomplished using test scripts, which are programs that test designers write to automate testing. These test scripts performs actions or mimicking user actions on GUI objects of the system to feed input data. Current approaches to load testing suffer from limitations. Their cost-effectiveness is highly dependent on the particular test scenarios that are used yet there is no support for choosing those scenarios. A poor choice of scenarios could lead to underestimating system response time thereby missing an opportunity to detect a performance [45].

1.2 Research Hypothesis

Our underlying research hypothesis is as follows:

> The use of metaheuristics and hybrid metaheuristics in combination with Q-learning can make it possible to automate the stress test execution process, improving the choice of new test cases for each interaction and finding scenarios that maximize the number of users of the application under test and minimize response time or find scenarios with a expected response time.

The purpose of this thesis is to show the validity of this hypothesis through the development of a testbed tool, algorithms that use hybrid metaheuristics and the Q-learning technique and application of validation experiments. This thesis will be useful for load test practitioners and software engineering researchers interested in large-scale testing software systems.

1.3 Thesis Outline

Figure 2.1 presents a brief summary of the thesis outline.

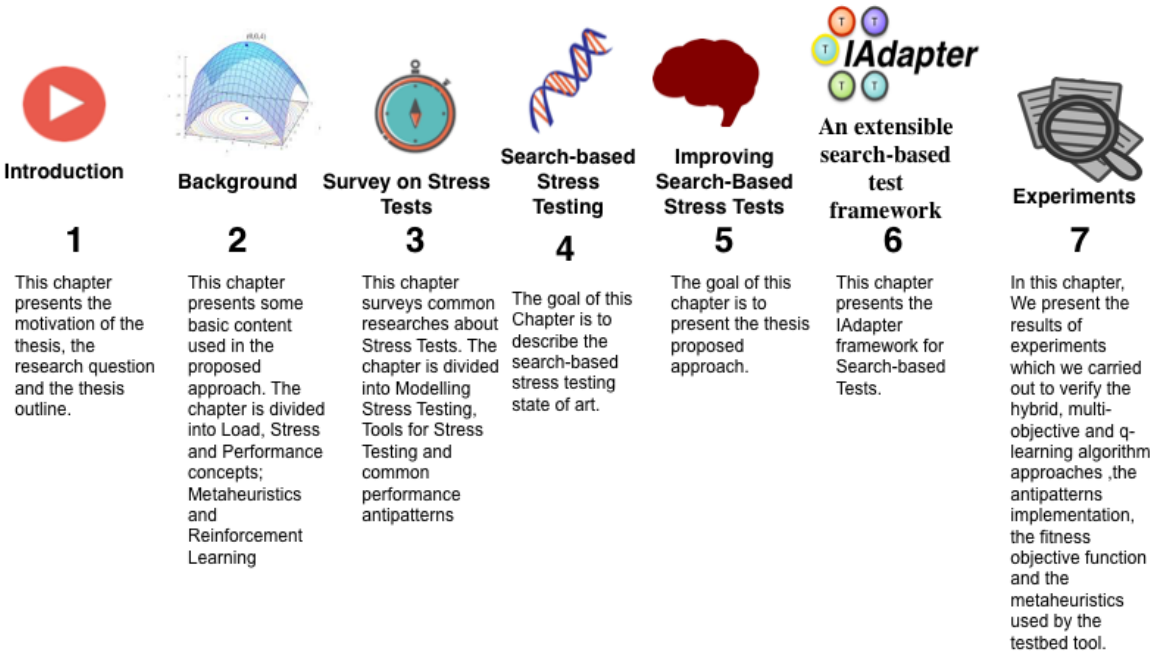


Figure 1.1: Thesis Outline

This thesis is organized as follows.

Chapter 3 reviews the main existing approaches in the research area referring to the search-based stress testing. In particular, two categories of approaches are out-lined: (i) Search-Based Stress Testing on Safety-critical systems; (ii) Search-Based Stress Testing on Industrial systems. In the context of the search-based process two metrics can be applied: processor cycles or response time.

Chapter 2

Background

This chapter presents a brief introduction of the basic concepts used in this thesis.

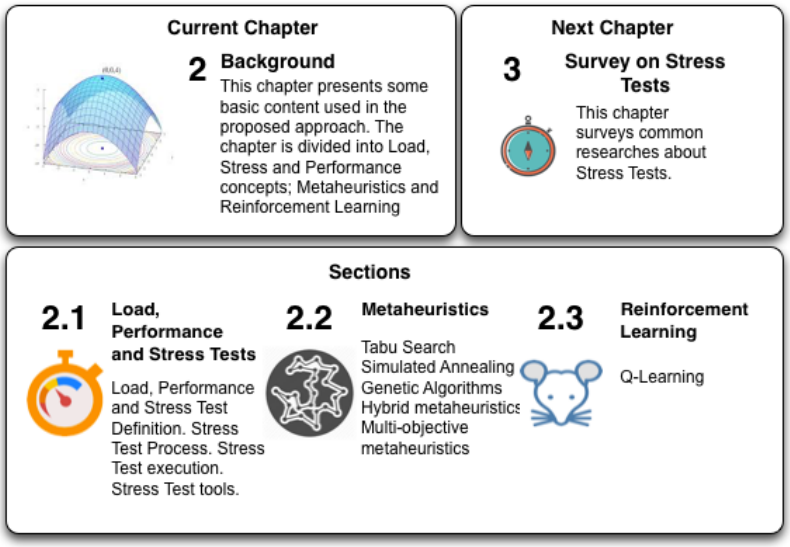


Figure 2.1: Chapter 2

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 [73].

Search-based testing (SBST) is the process of automatically generating tests according to a test adequacy criterion using search-based optimization algorithms, which are guided by a objective fitness function. The role of the fitness function is to capture a test objective that, when achieved, makes a contribution to the desired test adequacy criterion. SBST commonly uses metaheuristics as the search algorithm [49].

Simulated Annealing (SA) is a metaheuristic 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 [54]. Tabu Search (TS) is a metaheuristic that guides a local heuristic search procedure to explore the solution space beyond the local optimal and search with short term memory to avoid cycles [42].

Genetic Algorithms is a metaheuristic algorithm based on concepts adopted from genetic and evolutionary theories. A combination of one metaheuristic with components from other metaheuristics is called a hybrid metaheuristic. The concept of hybrid metaheuristics, combining different metaheuristic strategies and algorithms, dates back to the 1980s. Today, we can observe a generalized common agreement on the advantage of combining components from different search techniques and the tendency of designing hybrid techniques is widespread in the fields of operations research and artificial intelligence [73].

This thesis addresses the use of hybrid and multi-objective metaheuristics in conjunction with reinforcement learning techniques in search-based tests. Reinforcement learning (RL) refers to both a learning problem and a subfield of machine learning. As a learning problem, it refers to learning to control a system so as to maximize some numerical value which represents a long-term objective. The basic idea of Reinforcement learning is simply to capture the most important aspects of the real problem, facing a learning agent interacting with its environment to achieve a goal [87]. Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner needs to discover which actions yield the most reward by trying them [87].

2.1 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-related indicators to help stakeholders get informed about the quality of the application being tested [78] [23].

Typically, the most common kind of performance testing for Internet applications is load testing. Application load can be assessed in a variety of ways [69]:

- Concurrency. Concurrency testing seeks to validate the performance of an application with a given number of concurrent interactive users [69].
- Stress. Stress testing seeks to validate the performance of an application when certain aspects of the application are stretched to their maximum limits. This can include maximum number of users, and can also include maximizing table values and data values [69].
- Throughput. Throughput testing seeks to validate the number of transactions to be processed by an application during a given period of time. For example, one type of throughput test might be to attempt to process 100,000 transactions in one hour [69].

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 [29]. The purpose of this assessment is to demonstrate that the system reaches its performance objectives [78]. Term often used interchangeably with “stress” and “load” testing. Ideally “performance” testing is defined in requirements documentation or QA or Test Plans [58].

In a load testing, the system is evaluated at predefined load levels [29]. 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 [65]. A typical load test can last from several hours to a few days, during which system behavior data like execution logs and various metrics are collected [2].

Stress testing investigates the behavior of the system under conditions that overload its resources. The stress testing verifies the system behavior against heavy workloads [78] [58], 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 [29] [65].

The main difference between load tests, performance tests and stress tests are:

- Performance tests demonstrate that the system reaches its performance objectives.
- Load tests necessary use a load (concurrent or simultaneous users).
- Stress tests 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.

2.1.1 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. 2.2 shows a common Load, Performance and Stress test process [55].

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 [55].

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 consists of generating the load; and (3) Test Monitoring and Data Collection, which includes recording the system behavior during execution[55].

The core activities in conducting an usual Load, Performance and Stress tests are [33]:

- 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.

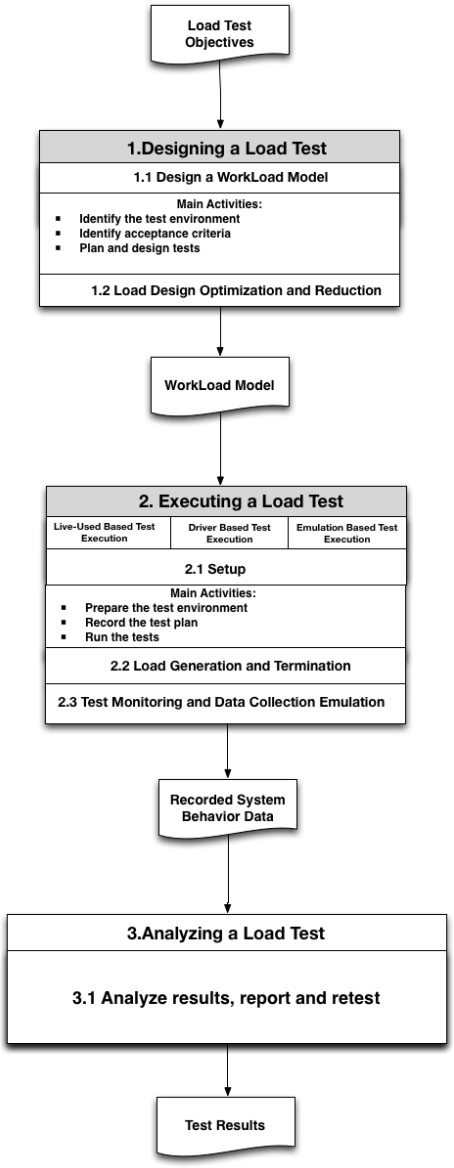


Figure 2.2: Load, Performance and Stress Test Process [55][33]

- 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 [23]. This task includes a description of the speed, availability, data volume throughput rate, response time, and recovery time of various

functions, stress, and so on. This serves as a basis for understanding the level of performance and stress testing that may be required to each test scenario [58].

- 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.

2.1.2 Stress Test Execution

The stress test execution consists of deploy the system and setup test execution ; generating the workloads according to the configurations and terminating the load when the load test is completed and recording the system behavior. There are three general approaches of load test executions [65][55]:

- Live-User Based Executions: The test examines a system's behavior when the system is simultaneously used by many users or execute a load test by employing a group of human testers.
- Driver Based Executions: The driver based execution approach automatically generate thousands or millions of concurrent requests for a long period of time using a software tool.
- Emulation Based Executions: The emulation based load test execution approach performs the load testing on special platforms and doesn't require a fully functional system and conduct load testing.

Usually, a stress test execution it is performed with Driver Based Executions approach [33] [64] [95]. There are three categories of load drivers [55]:

- Benchmark Suite: specialized load driver, designed for one type of system. For example, LoadGen is a load driver specified used to load test the Microsoft Exchange MailServer.
- Centralized Load Drivers: refer to a single load driver, which generates the load.
- Peer-to-peer Load Drivers: refer to a set of load drivers, which collectively generate the target testing load. Peer-to-peer load drivers usually have a con- troller component, which coordinates the load generation among the peer load drivers.

A stress test need to perform hundreds or thousands of concurrent requests to the application under test. 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.

Stress Testing Tools

There are several tools to execution of stress testing. Stress testing tools are software products based on workload models to generate request sequences similar to real requests. They are designed and implemented as versatile software tools for performing tuning or capacity planning studies. Ussualy, the tool functions is semi-automated, whereas the execution of the tests itself is performed by a tool, the choice of scenarios to be executed as well as the decision to start new execution batteries are activities of the test designer or tester. Normally, load test tools uses test scripts. Test scripts are written in a GUI testing framework or a backend server-directed performance tool such as JMeter. These frameworks are the basis on which performance testing is mostly done in industry. Performance test scripts imitate large numbers of users to create a significant load on the application under tests. Stress testing tools typically have the following components [45] [65]:

- 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.

Comparing stress test tools is a laborious and difficult task since they offer a large amount and diversity of features [31]. In next subsection we contrast stress testing tools according to a wide set of features and capabilities, focusing on their ability to realize search-based tests or have learning capacities.The stress test tools was categorized in three different groups [64]:

- Benchmarks that model the client and server paradigm in Web context.
- Software products to evaluate performance and functionality of a given Web application, such as LoadRunner, WebLOAD and JMeter.
- Testing tools and other approaches for traffic generation based on HTTP traces.

Penã-Ortiz et al. present a study where stress test tools are compared using 12 features [64]:

- Distributed architecture. This refers to the ability to distribute the generation process among different nodes.
- Analytical-based architecture. This feature represents the capability to use analytical and mathematical models to define the workload.
- Business-based architecture. When defining a testing environment, the simulator architecture should implement the same features as the real environment.

Table 2.1: Benchmarks group

Feature/Tool	WebStone	SpecWeb	SURGE	Web Polygraph	TPC-W
Analytical-Based Architecture	Full support	Full support	Full support	Full support	Full support
Distributed Architecture	Full support	Full support	Full support	Full support	
Business-Based Architecture		Partial Suport		Partial Suport	Full support
Client Parameterization	Full support	Full support		Partial Suport	Full support
Workload Types		Full support			Full support
Functional Testing					
LAN and WAN					
Multi-platform	Full support	Full support	Full support	Full support	Full support
Ease of Use					
Performance Reports	Partial Suport	Full support		Full support	Full support
Open Source	Full support		Full support	Partial Suport	Full support
User's Dynamism					Partial Suport

- Client parameterization. This is the ability to parameterize generator nodes.
- Workload types. Some generators organize the workload in categories or types.
- Testing the Web application functionality (functional testing).
- Multiplatform refers to a software package that is implemented in multiple types of computer platforms, interoperating among them.
- Differences between LAN and WAN.
- The generator should be a friendly application.
- The load test tool has performance reports.
- The load test tool is open-source.
- Users' dynamism. The users has the hability of change they behaviour in during the test.

WebStone was designed by Silicon Graphics in 1996 to measure the performance of Web server software and hardware products. Nowadays, both executable and source actualized code for WebStone are available for free. The benchmark generates a Web server load by simulating multiple Web clients navigating a website. All the testing done by the benchmark is controlled by a Webmaster, which is a program that can be run on one of the client computers or on a different one [64] [92] .

TPC Benchmarkt (TPC-W) is a transactional Web benchmark defined by the Transaction Processing Performance Council that models a representative e-commerce evaluating the architecture performance on a generic profile.The models uses a remote browser emulator to generate requests to server under test. TPC-W adopts the CBMG model to define the workloads in spite of this model only characterizing user dynamic

Table 2.2: My caption

Feature/Tool	LoadRunner	WebLOAD	JMeter
Analytical-Based Architecture	Partial Suport	Partial Suport	Partial Suport
Distributed Architecture	Full support	Full support	Full support
Business-Based Architecture	Full support	Full support	Full support
Client Parameterization	Full support	Full support	Full support
Workload Types	Full support	Full support	Partial Suport
Functional Testing	Full support	Full support	Partial Suport
LAN and WAN			
Multi-platform	Full support	Full support	Full support
Ease of Use	Full support	Full support	Partial Suport
Performance Reports	Partial Suport	Full support	Full support
Open Source			Partial Suport
User's Dynamism	Partial Suport	Partial Suport	Partial Suport

behavior partially. The remote browser emulators are located in the client side and generate workload towards the e-commerce Web application, which is located in the server side (e-commerce server) [64] [63].

Open STA is an open source software developed in C++, and released under the GPL licence. OpenSTA provides a script language which permits to simulate the activity of a user. This language can describe HTTP/S scenario and all the test executions is managed in a graphical interface. The composition of the test is very simple, allowing the tester choose scripts for a test and a remote computer that will execute each test.

LoadRunner is one of the most popular industry-standard software products for functional and performance testing. It was originally developed by Mercury Interactive, but nowadays it is commercialized by Hewlett-Packard. LoadRunner supports the definition of user navigations, which are represented using a scripting language. The basic steps are recorded, creating a shell script. Next, this script is then taken offline, and undergoes further manual steps such as data parameterization and correlations. Finally, the desired performance scripts are obtained after adding transactions and any other required logic (Fig. 2.4). LoadRunner scripting only permits partial reproduction of user dynamism when generating Web workload, because it cannot define either advanced interactions of users, such as parallel browsing behavior, or continuous changes in user's behaviors [64].

WebLOAD is a software tool for Web performance commercialized by RadView. It is oriented to explore the performance of critical Web applications by quantifying the utilization of the main server resources. The tool creates scenarios that try to mimic the navigations of real users. To this end, it provides facilities to record, edit and debug test scripts, which are used to define the scenarios on workload characterization. The execution environment is a console to manage test execution, whose results are analyzed in the Analytics application. Since WebLOAD is a distributed system, it is possible to deploy several load generators to reproduce the desired load. Load generators can also be used as probing clients where a single virtual user is simulated to evaluate specific statistics of a single user. These probing clients resemble the experience of a real user using the system while it is under load [64].

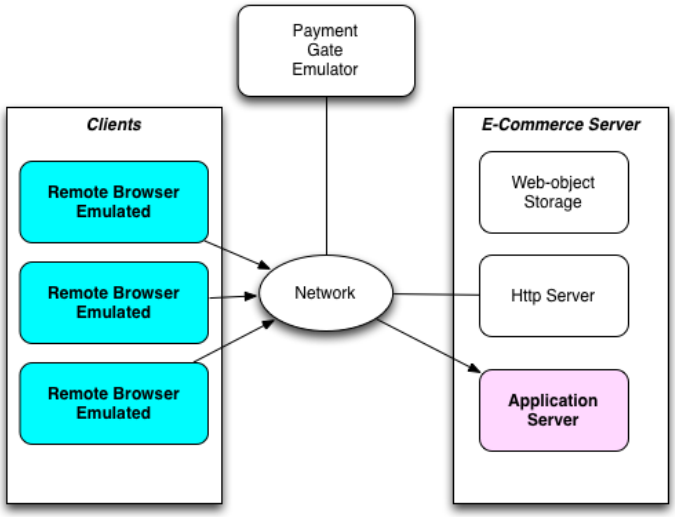


Figure 2.3: TPC-W architecture [64] [63]

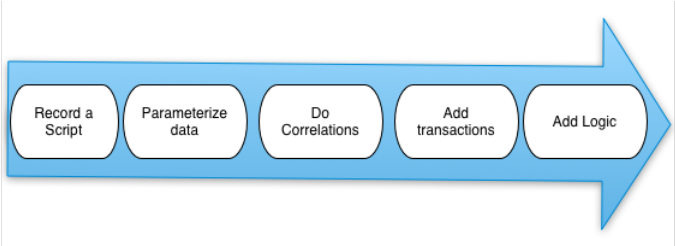


Figure 2.4: Load Runner Scripting

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 [48] [33].

Figures **??**,**??** and **??** summarizes the studied workloads generators as well as the grade (full or partial) in which they fulfill the features described bellow. None of the presented tools uses heuristic or learning resources when choosing the scenarios to be tested or the workloads to be applied in the test.

Apache JMeter

Apache JMeter is user friendly and a flexible open source solution for performance verification. Apache JMeter is designed in pure Java application. It is used to generate heavy loads on the servers or objects to test its strength or analyze overall performance under different load types. To briefly explain the solution how it works, as follows: An Regular Expression Extractor captures the dynamic values as mentioned above and

stored in a temporary variable. The values which has been extracted and stored in temporary variables are subsequently utilized by immediate requests/re directions using HTTP samplers [**?**]. 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.

2.2 Metaheuristics

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 [17]:

- 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 (Trajectory methods). 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.

2.2.1 Trajectory methods

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

Neighborhood

The definition of Neighborhood is a required common step for the design of any Single-Solution metaheuristic (S-metaheuristic). The neighborhood structure it is a important piece in the performance of an S-metaheuristic. If the neighborhood structure is not adequate to the problem, any S-metaheuristic will fail to solve the problem. The neighborhood function N is a mapping: $N : S \rightarrow N^2$ that assigns to each solution s of S a set of solutions $N(s) \subset$ S [89].

The neighborhood definition depends representation associated with the problem. For permutation-based representations, a usual neighborhood is based on the swap operator that consists in swapping the location of two elements s_i and s_j of the permutation [89]. The Fig. 2.5 presents a example where a set of neighbors is found by permutation.

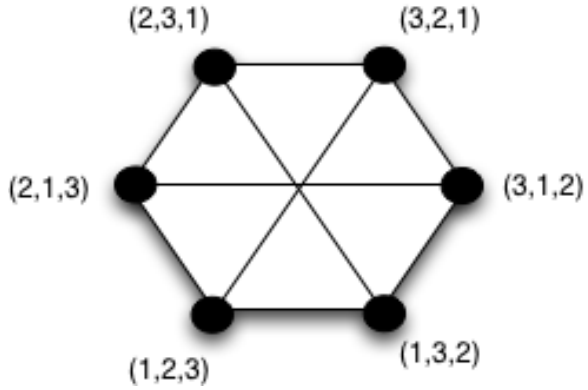


Figure 2.5: An example of neighborhood for a permutation [89].

Single-Solution Based Metaheuristics methods are characterized by a trajectory in the search space. Two common S-metaheuristics methods are Simulated Annealing and Tabu Search.

Simulated Annealing

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 [73].

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
```

Algorithm 2 Tabu Search Algorithm

```
    s ← GenerateInitialSolution()
2:  InitializeTabuLists(TL1,...,TLr)
    while termination conditions not met do
4:      Na(s) ← {s1 ∈ N(s)|s1 does not violate a tabu condition, or it satisfies at least one aspiration condition }
        s1 ← argmin{f(s2)|s2 ∈ Na(s)}
6:      UpdateTabuLists(TL1,...,TLr,s,s1)
        s ← s1
8:  end while
```

Tabu Search

Tabu Search uses a tabu list to keep track of the last moves, and don't allow going back to these [42]. 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.

2.2.2 Population-based metaheuristics

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 [88].

Population-based metaheuristics are comprised of several components [52] [81] :

- a representation of the solution, refered as the chromosome;
- fitness of each chromosome, refered as objective function;
- the genetic operations of crossover and mutation which generate new offspring.

The crossover operation or recombination recombines two or more individuals to produce new individuals. Mutation or modification operators causes a self-adaptation of individuals [17]. In Search-based tests, the crossover operator creates two new test cases T1' and T2' by combining test cases from two pre-existing test cases T1 and T2 [5]. 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 [73].

Algorithm 3 Genetic Algorithm

1: $s \leftarrow GenerateInitialSolution()$
2: Evaluate(P)
3: **while** termination conditions not met **do**
4: $P_1 \leftarrow Recombine(P)$
5: $P_2 \leftarrow Mutate(P_1)$
6: $Evaluate(P_2)$
7: $P \leftarrow Select(P_2, P)$
8: **end while**

2.2.3 Hybrid Metaheuristics

A combination of one metaheuristic with components from other metaheuristics is called a hybrid metaheuristic. The concept of hybrid metaheuristics has been commonly accepted only in recent years, even if the idea of combining different metaheuristic strategies and algorithms dates back to the 1980s. Today, we can observe a generalized common agreement on the advantage of combining components from different search techniques and the tendency of designing hybrid techniques is widespread in the fields of operations research and artificial intelligence [73].

There are two main categories of metaheuristic combinations: collaborative combinations and integrative combinations. These are presented in Fig. 2.6 [74].

Collaborative combinations use an approach where the algorithms exchange information, but are not part of each other. In this approach, algorithms may be executed sequentially or in parallel.

One of the most popular ways of metaheuristic hybridization consists in the use of trajectory methods inside population-based methods. Population-based methods are better in identifying promising areas in the

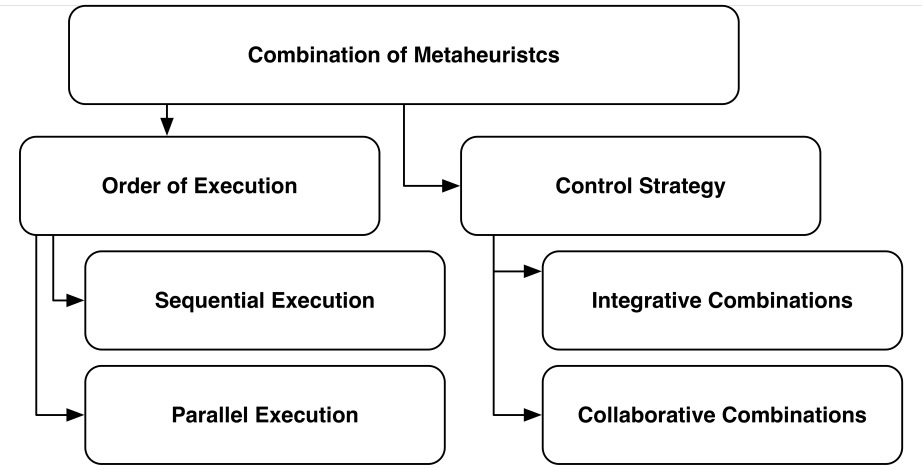


Figure 2.6: Categories of metaheuristc combinations [71]

search space from which trajectory methods can quickly reach good local optima. Therefore, metaheuristic hybrids that can effectively combine the strengths of both population-based methods and trajectory methods are often very successful [73].

2.2.4 Multi-objective heuristics

NSGA-II Multi-objective heuristics

Many real optimization problems require optimizing multiple conflicting objectives with each other. There is no single optimal solution, but a set of alternative solutions. The objectives that have to be optimized are often in competition with one another and may be contradictory; we may find ourselves trying to balance the different optimization objectives of several different goals [49] [**?**]. The image of all the efficient solutions is called the Pareto front or Pareto curve or surface. The shape of the Pareto surface indicates the nature of the trade-off between the different objective functions. An example of a Pareto curve is reported in Fig. 2.7. Multi-objective optimization methods have as main purposes to minimize the distance between the non-dominated front and the Pareto optimal front and find a set of solutions that are as diverse as possible.

What distinguishes multi-objective evolutionary algorithms from single objective metaheuristics is how they rank and select individuals in the population. If there is only one objective, individuals are naturally ranked according to this objective, and it is clear which individuals are best and should be selected as parents. In the case of multiple objectives, it is still necessary to rank the individuals, but it is no longer obvious how to do this. Most people probably agree that a good approximation to the Pareto front is characterized by:

- a small distance of the solutions to the true Pareto frontier,
- a wide range of solutions, i.e., an approximation of the extreme values, and

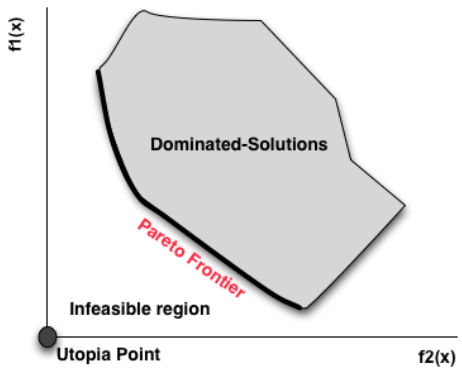


Figure 2.7: An optimized Pareto front example

- a good distribution of solutions, i.e., an even spread along the Pareto frontier.

Multi-objective metaheuristics rank individuals according to the goals mentioned above. The implementation adapted for this paper is based on the non-dominated Sorting Genetic Algorithm II (NSGA-II) algorithm. Deb et al. proposed the NSGA-II, taking into account the need to reduce computational complexity in non-dominated classification, while introducing elitism and eliminating subjectivity in the allocation of the sharing parameter [**?**]. NSGA-II is a multi-objective algorithm, based on GAs, and implements the concept of dominance, in other words, to classify the total population in fronts according to the degree of dominance. According to NSGA-II, the individuals that are located on the first front are considered the best solutions of that generation, while in the last front are the worst. Using this concept, one can find more consistent results, located closer to the Pareto region, and that are better adapted to the type of problem.

The NSGA algorithm II applies a fitness evaluation in an initial population (Figure 2.8- ❶ and ❷). The populations are ranked using multiple tournament selections, which consist of comparing two solutions (Figure 2.8- ❸). In order to estimate the density of the solutions surrounding a particular solution in the population, the common distance between the previous solution and the posterior is calculated for each of the objectives. This distance serves as an estimate of the size of the largest cuboid that includes solution i without including any other solution of the population. A solution i beats another solution if:

- Solution i has a better rank, then $Rank_i < Rank_j$.
- Both solutions have the same rank, but i has a greater Distance than j, then $Rank_i = Rank_j$ and $Distance_i > Distance_j$.

At the end of each analysis a certain group of individuals are classified as belonging to a specific category called the front, and upon completion of the classification process, all individuals will be inserted into one of the n fronts. Front 1 is made up of all non-dominated solutions. Front 2 can be achieved by considering all non-dominated solutions excluding solutions from front 1. For the determination of front 3, solutions

previously classified on front 1 and 2 are excluded, and so on until all individuals have been classified on some front.

After selection, recombination and mutation are performed as in conventional GAs (Figure 2.8- ❹). The two sets (father and son of the same dimension) are united in a single population (dimension 2) and the classification is applied in dominance fronts. In this way, elitism is guaranteed preserving the best solutions (fronts are not dominated) in the latest population (Figure 2.8- ❻).

However, not all fronts can be included in the new population. Thus, Deb et al. proposed a method called crowd distance, which combines the fronts not included in the set, to compose of the last spaces of the current population, guaranteeing the diversity of the population [**?**]. The NSGA-II algorithm creates a set of front lines, in which each front containing only non–dominating solutions. Within a front, individuals are rewarded for being 'spread out'. The algorithm also ensures that the lowest ranked individual of a front still has a greater fitness value than the highest ranked individual of the next front [**?**].

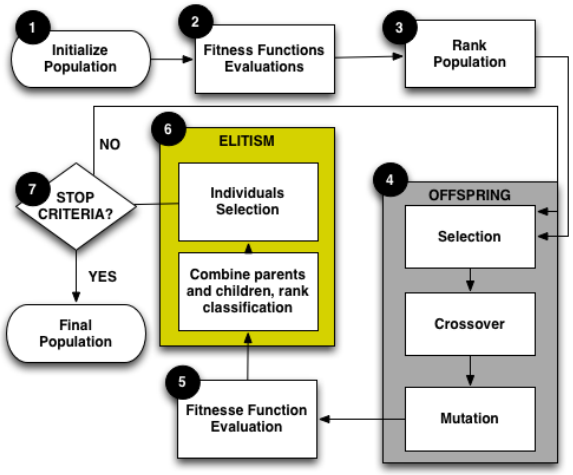


Figure 2.8: NSGA-II Algorithm

2.3 Reinforcement Learning

Reinforcement learning (RL) refers to both a learning problem and a subfield of machine learning. As a learning problem, it refers to learning to control a system so as to maximize some numerical value which represents a long-term objective. The basic idea of Reinforcement learning is simply to capture the most important aspects of the real problem, facing a learning agent interacting with its environment to achieve a goal [87]. Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner needs to discover which actions yield the most reward by trying them [87].

A typical setting where reinforcement learning operates is shown in Figure 2.9: A controller receives the

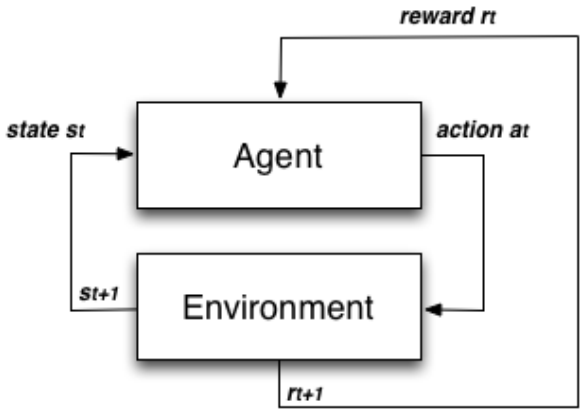


Figure 2.9: Example of interation between some agent and the environment

controlled system's state and a reward associated with the last state transition. It then calculates an action which is sent back to the system.

In Reinforcement Learning, an agent wanders in an unknown environment and tries to maximize its long term return by performing actions and receiving rewards. The challenge is to understand how a current action will affect future rewards. A good way to model this task is with Markov Decision Processes (MDP). Markov decision processes (MDPs) provide a mathematical framework for modeling decision making. In Reinforcement Learning, all agents act in two phases: Exploration vs Explotation. In Exploration phase, the agents try to discover better action selections to improve its knowledge. In Exploitation phase, the agents try to maximize its reward, based on what is already know.

One of the challenges that arise from reinforcement learning is the trade-off between exploration and exploitation. To obtain a large reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to exploit what it already knows in order to obtain a reward, but it also has to explore in order to make better action selections in the future.

Q-learning is a model-free reinforcement learning technique. Q-learning is a multiagent learning algorithm that learns equilibrium policies in Markov games, just as Q-learning learns to optimize policies in Markov decision processes [46].

Q-learning and related algorithms try to learn the optimal policy from its history of interaction with the environment. A history of an agent is a sequence of state-action-rewards.Where s_n is a state, a_n is an action and r_n is a reward:

$$< s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, a_3, r_4, s_4.... >, \tag{2.1}$$

In Q-Learning, the system's objective is to learn a control policy $\pi = \sum_{n=0}^{\infty} \gamma^n r_t + n$, where π is the discounted cumulative reward, γ is the discount rate (01) and r_t is the reward received after the execution of

an action at time t. Figure 2.10 shows the summary version of Q-Learning algorithm. The first step is to generate the initial state of the MDP. The second step is to choose the best action or a random action based on the reward, hence the actions with best rewards are chosen.

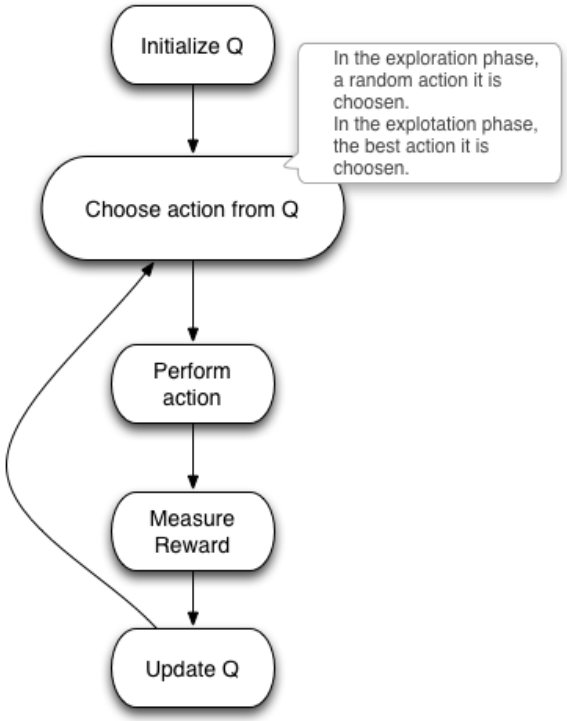


Figure 2.10: Q Learning algorithm

Chapter 3

A Survey on Stress Testing Software Systems

This chapter surveys the state of the art literature in stress testing research. The thesis extends the survey presented by Jiang et al. [55] and Afzal et al. [2] to the Stress Testing context . This survey will be useful for stress testing practitioners and software engineering researchers with interests in testing and analyzing software systems. The paper use the systematic review method proposed by Kitchenham [**?**].

The aim of a systematic review is to find as many primary studies relating to the research question as possible using an unbiased search strategy. The rigour of the search process is one factor that distinguishes systematic reviews from traditional reviews [**?**]. The systematic review is based on a comprehensive set of 97 articles obtained after a multi-stage selection process and have been published in the time span 1994–2016.

3.0.1 Planning a Systematic Review

A systematic review of the literature details a protocol describing the process and the methods to be applied. The most important activity during the planning phase is the formulation of research questions. To Kitchenham, before undertaking a systematic review researchers must ensure that it is necessary and the protocol should be able to answer some questions [**?**]:

- What are the objectives of this review?
- What sources were searched to identify primary studies? Were there any restrictions?
- What were the criteria for inclusion / exclusion and how they are applied?
- What criteria were used to evaluate the quality of the primary studies?
- How were the quality criteria applied?

- How was the data extracted from primary studies?
- What were the differences between studies investigated?
- Because the data were combined?

3.0.2 Research Questions

In order to examine the evidence of stress testing properties, we proposed the following two research questions:

- How modeling a stress test?
- What are the main problems found by stress tests?

3.0.3 Generation of search strategy

The population in this study is the domain of software testing. Intervention includes application of stress test techniques to test different types of non-functional properties. The primary studies used in this review were obtained from searching databases of peer-reviewed software engineering research that met the following criteria:

- Contains peer-reviewed software engineering journals articles, conference proceedings, and book chapters.
- Contains multiple journals and conference proceedings, which include volumes that range from 1996 to 2017.
- Used in other software engineering systematic reviews.

The resulting list of databases was:

- ACM Digital Library
- Google Scholar
- IEEE Electronic Library
- Inspec
- Scirus (Elsevier)
- SpringerLink

The search strategy was based on the following steps:

- Identification of alternate words and synonyms for terms used in the research questions. This is done to minimize the effect of differences in terminologies.
- Identify common stress testing properties for searching.
- Use of Boolean OR to join alternate words and synonyms.
- Use of Boolean AND to join major terms

We used the following search terms:

- Load Testing: load test, Load Testing
- Stress Testing: stress test, stress testing
- Performance Testing: performance tests
- Test tools: jmeter, load runner, performance tester

3.0.4 Study selection criteria and procedures for including and excluding primary studies

The idealized selection process was done in two parts: an initial document selection of the results that could reasonably satisfy the selection criteria based on a title and the articles abstract reading, followed by a final selection of the initially selected papers based on the introduction and conclusion reading of the papers. The following exclusion criteria is applicable in this review, i.e. exclude studies that:

- Do not relate to stress testing.
- Do not relate to load testing tool.
- Do not relate to load/stress testing model.

From 366 initial papers, 97 papers was selected.

3.0.5 Data Synthesis

Data synthesis involves collating and summarising the results of the included primary studies. Synthesis can be descriptive (non-quantitative). The studies was categorized by:

- Type of stress test properties;
- Type of research paper (Thesis, Journal Article, Conference Paper, Book Section or Book)
- Methodology used by the test (Model based Test, FOREPOST, Search-based Tests)

3.1 Research Question 1:How is a proper stress designed?

The design of a stress test depends intrinsically on the load model applied to the software under test. Based on the objectives, there are two general schools of thought for designing a proper load to achieve such objectives [2]:

- Designing Realistic Loads (Descriptive Workload).
- Designing Fault-Inducing Loads (Generative Workload).

In Designing Realistic Loads, the main goal of testing is to ensure that the system can function correctly once. Designing Fault-Inducing Loads aims to design loads, which are likely to cause functional or non-functional problems [2].

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 [65].

The term workload represents the size of the demand that will be imposed on the application under test in an execution. The metric used to 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 [35] [65] [44].

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 [35] [65]. 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 [35] [65].

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 [35].

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. This model is applied to all the workload attributes, e.g., computation, memory usage, I/O behavior, communication, etc. [35]. Fig. 3.1 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.

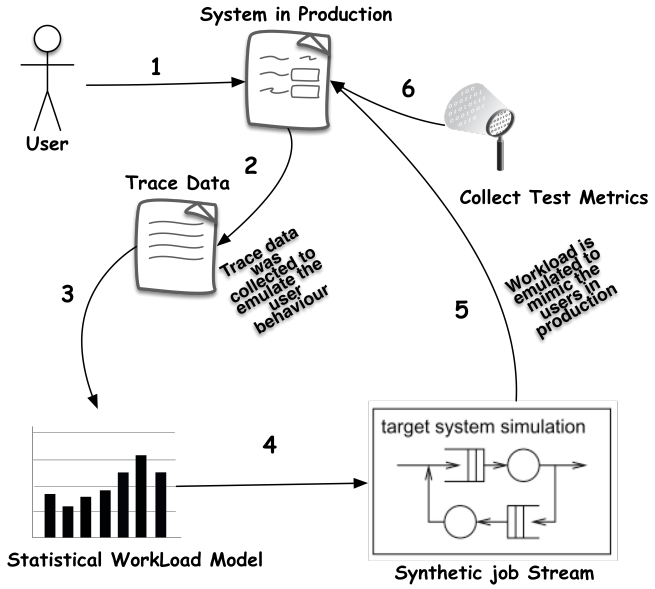


Figure 3.1: Workload modeling based on statistical data [29]

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 [35]. 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. 3.2).

Both load models 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, where the resulting loads usually only cover a small portion of the testing objectives [55]. The presented research work uses a generative model.

There are several approaches to design generative or descriptive workloads:

- Model-based Stress testing: a usage model is proposed to simulate users' behaviors.
- Feedback-Directed Learning Software Testing: is an adaptive, feedback-directed learning testing system that learns rules from system execution [59] [100].

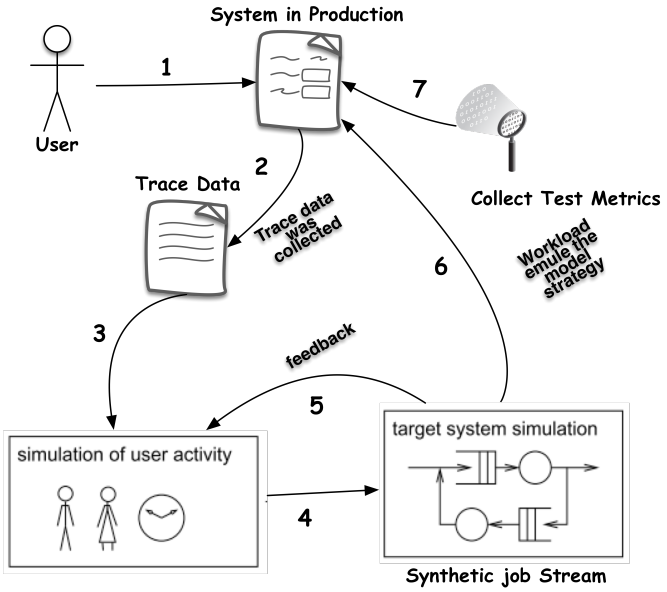


Figure 3.2: Workload modeling based on the generative model [29]

- Search-based Stress testing.

Search-Based Stress testing will be detail explained in the chapter 3.

3.1.1 Model-based Stress Testing

Model-based testing is an application of models to represent the desired behavior of a System Under Test or to represent testing strategies in a test. Some research aproaches proposes models to simulate or generate realistic loads. Model-based testing (MBT) is a variant of testing that relies on explicit behaviour models that encode the intended behaviours of a system under test. Test cases are generated from one of these models or their combination [62] [1].

The model paradgim is what paradigm and notation are used to describe the model. There are many different modelling notations that have been used for modelling the behaviour of systems for test generation purposes [62] [51].

- State-Based (or Pre/Post) Notations. These model a system as a collection of variables, which represent a snapshot of the internal state of the system, plus some operations that modify those variables. Each operation is usually defined by a precondition and a postcondition, or the postcondition may be written as explicit code that updates the state [62].

- Transition-based Notations. These focus on describing the transitions between different states of the

system. Typically, they are graphical node-and-arc notations, like finite state machines (FSMs). Examples of transition-based notations used for MBT include FSMs themselves, statecharts, labelled transition systems and I/O automata [62].

- History-based Notations. These notations model a system by describing the allowable traces of its behaviour over time. Message-sequence charts and related formalisms are also included in this group. These are graphical and textual notations for specifying sequences of interactions between components [62].

- Functional Notations. These describe a system as a collection of mathematical functions. The functions may be first-order only, as in the case of algebraic specifications, or higher-order, as in notations like HOL [62].

- Operational Notations. These describe a system as a collection of executable processes, executing in parallel. They are particularly suited to describing distributed systems and communications protocols. Examples include process algebras such as CSP or CCS as well as Petri net notations. Slightly stretching this category, hardware description languages like VHDL or Verilog are also included in this category [62].

- Stochastic Notations. These describe a system by a probabilistic model of the events and input values and tend to be used to model environments rather than SUTs. For example, Markov chains are used to model expected usage profiles, so that the generated tests scenarios [62].

- Data-Flow Notations. These notations concentrate on the data rather than the control flow. Prominent examples are Lustre, and the block diagrams of Matlab Simulink, which are often used to model continuous systems [62].

Table presents the papers found by the survey. All results was classified by model and paradigm of the model-based test.

A User Community Modeling Language (UCML) is a set of symbols that can be used to create visual system usage models and depict associated parameters [95]. The Fig. 3.3 shows a sample where all users realize a login into the application unde test. Once logged in, 40% of the users navigates on the aplication, 30% of the users realizes downloads. 20% of users realizes uploads and 10% of users performs deletions.

Another technique to create workload models it is Stochastic Formcharts. The work of Draheim and Weber's Formoriented analysis is a methodology for the specification of ultra-thin client based systems. Formoriented models describe a web application as a bipartite state machine which consists of pages, actions, and transitions between them. Stochastic Formcharts are the combination of formoriented model and probability features. The Fig. 3.4 shows a sample where all users have a probability of 100% of realize a login into the application under test. Once logged in, users have a probability of 40% of navigate on the application and so on [30].

Table 3.1: My caption

Model	Paper	Paradigm	Year
BeliefDesire-Intention	[8]	Operational	2016
Markov-Chains	[12]	Transition-based	1995
	[16]	Transition-based	2007
	[14]	Transition-based	1994
	[13]	Transition-based	1993
MCM Model	[101]	State-based	2010
Other Aproaches	[32]	UPPAAL model checker	2013
	[10]	Model Decomposition	2015
Petri Nets	[20]	Operational	2009
Stochastic Form Model	[21]	Transition-based	2007
	[30]	Transition-based	2006
State-Machine Models	[34]	State-based	2012
	[85]	State-based	2013
	[36]	Transition-based	2016
	[77]	-	2016
	[9]	Transition-based	2014
	[41]	Transition-based	2016
	[50]	Transition-based	2007
	[51]	Transition-based	2009
	[**?**]	State-based	2016
Symbolic Transition System	[7]	Transition-based	2013
Timed Automata	[6]	Transition-based	2013
UCML	[15]	Transition-based	1999
UML	[102]	Functional	2007
	[80]	Functional	2013
	[75]	Functional	2009
	[79]	Functional	2013
	[67]	Functional	2014
	[56]	Functional	2005
	[76]	Functional	2014
	[25]	Functional	2011
	[57]	Functional	2007

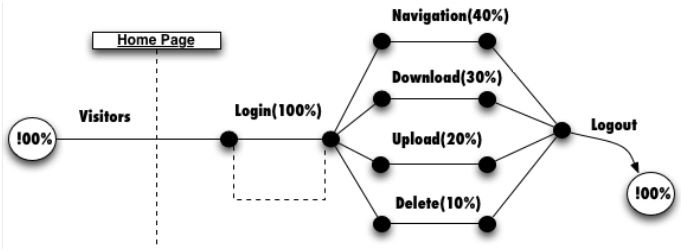


Figure 3.3: User community modeling language [95]

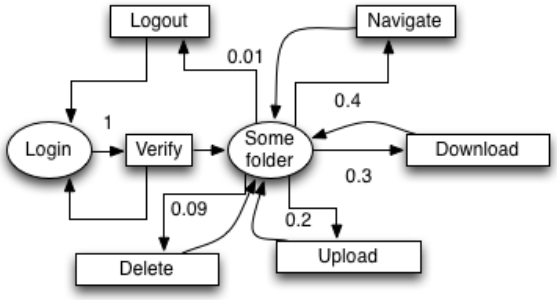


Figure 3.4: Stochastic Formcharts Example [30] [95]

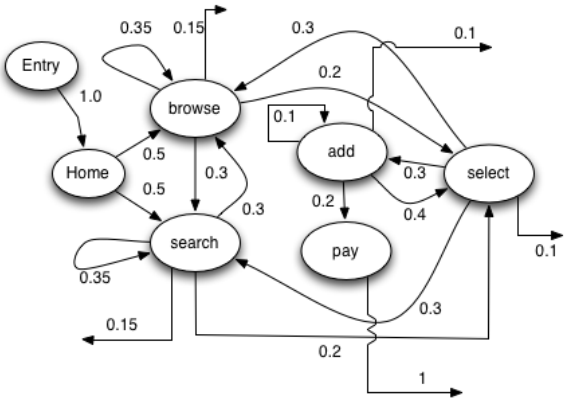


Figure 3.5: Example of a Customer Behavior Model Graph (CBMG) [63] [55] [64]

One way to capture the navigational pattern within a session is through the Customer Behavior Model Graph (CBMG). Figure 3.5 depicts an example of a CBMG showing that customers may be in several different states—Home, Browse, Search, Select, Add, and Pay—and they may transition between these states as indicated by the arcs connecting them. The numbers on the arcs represent transition probabilities. A state not explicitly represented in the figure is the Exit state [63] [55] [64].

Garousi et al. proposes derivate Stress Test Requirements from an UML model. The input model consists of a number of UML diagrams. Some of them are standard in mainstream development methodologies and others are needed to describe the distributed architecture of the system under test (Fig. 3.6).

Vogele et al. presents an approach that aims to automate the extraction and transformation of workload specifications for an model-based performance prediction of session-based application systems. The research also presents transformations to the common load testing tool Apache JMeter and to the Palladio Component Model [94]. The workload specification formalism (Workload Model) consists of the following components, which are detailed below and illustrated in Fig. 3.7:

- An Application Model, specifying allowed sequences of service invocations and SUT-specific details for generating valid requests.
- A set of Behavior Models,each providing a probabilistic representation of user sessions in terms of invoked services.
- A Behavior Mix, specified as probabilities for the individual Behavior Models to occur during workload gen- eration.
- A Workload Intensity that includes a function which specifies the number of concurrent users during the workload generation execution.

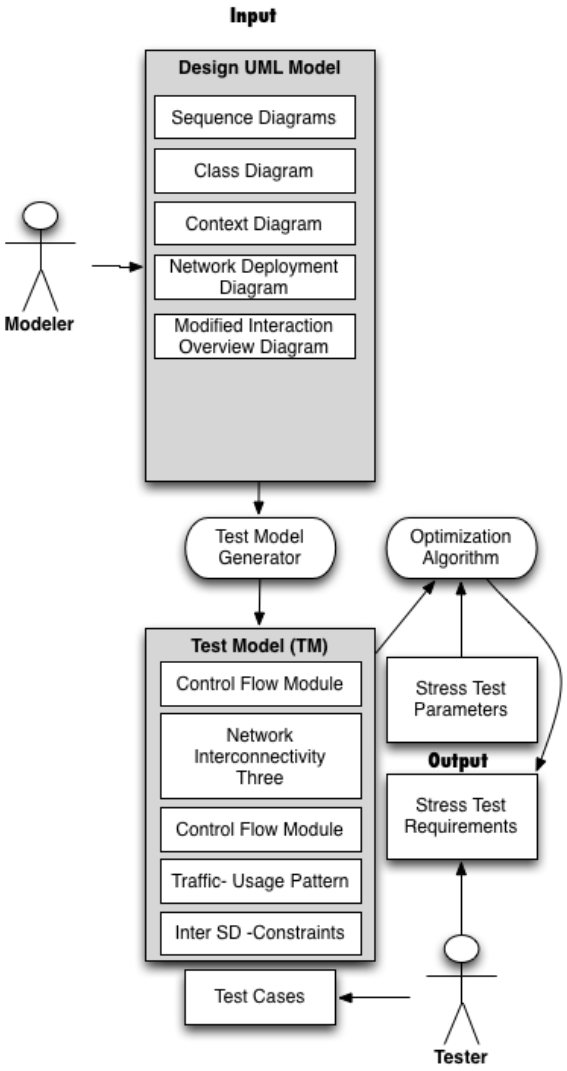


Figure 3.6: Model-based stress test methodology

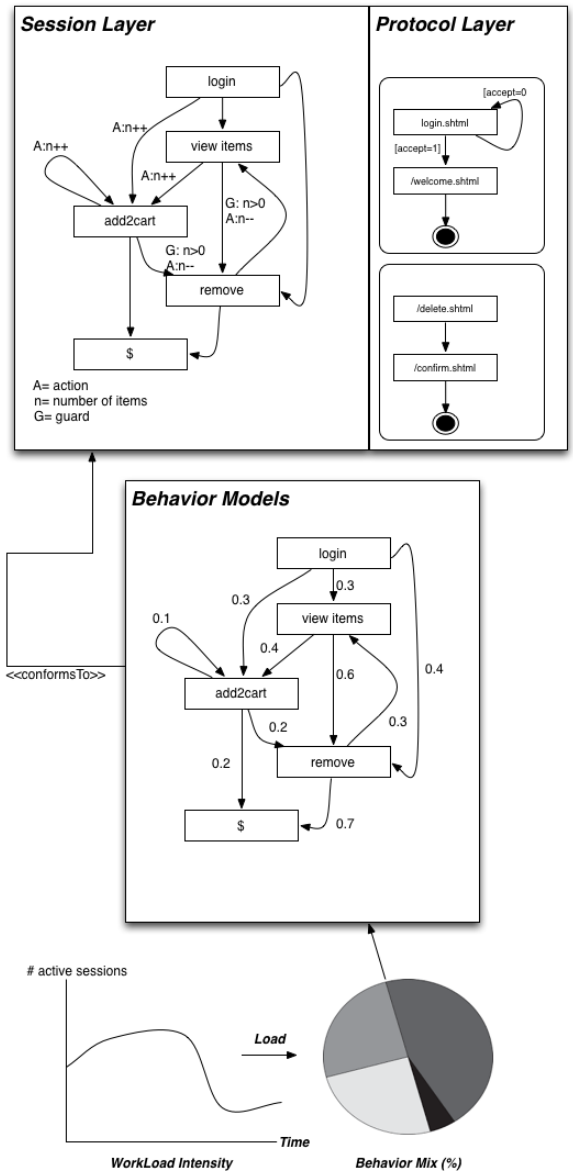


Figure 3.7: Exemplary workload model

3.1.2 Feedback-ORiEnted PerfOrmance Software Testing

Feedback-ORiEnted PerfOrmance Software Testing (FOREPOST) is an adaptive, feedback-directed learning testing system that learns rules from system execution traces and uses these learned rules to select test input data automatically to find more performance problems in applications when compared to exploratory random performance testing [45].

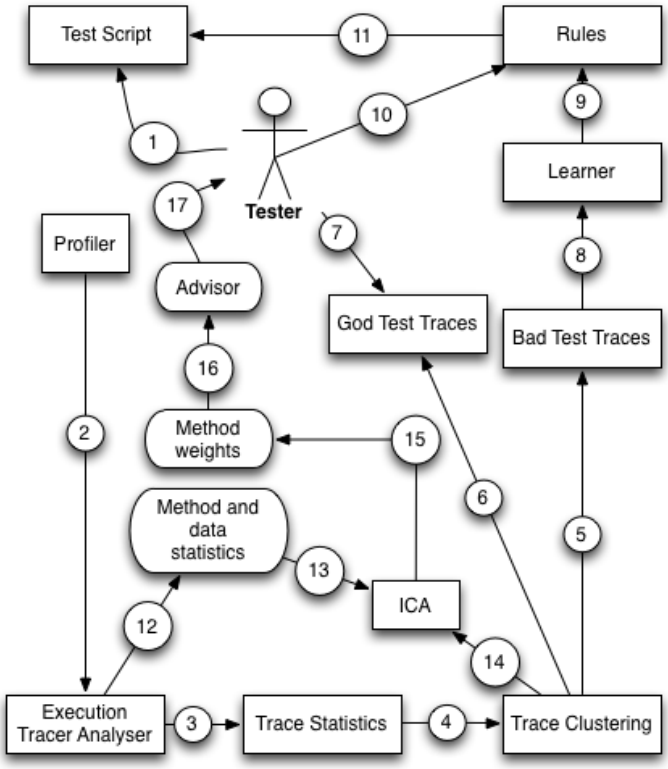


Figure 3.8: The architecture and workflow of FOREPOST

FOREPOST uses runtime monitoring for a short duration of testing together with machine learning techniques and automated test scripts to reduce large amounts of performance-related information collected during AUT runs to a small number of descriptive rules that provide insights into properties of test input data that lead to increased computational loads of applications.

The Fig. 3.8 presents the main workflow of FOREPOST solution. The first step, The Test Script is written by the test engineer(1). Once the test script starts, its execution traces are collected (2) by the Profiler, and these traces are forwarded to the Execution Trace Analyzer, which produces (3) the Trace Statistics. The trace statistics is supplied (4) to Trace Clustering, which uses an ML algorithm, JRip to perform unsupervised clustering of these traces into two groups that correspond to (6) Good and (5) Bad test traces.

The user can review the results of clustering (7). These clustered traces are supplied (8) to the Learner that uses them to learn the classification model and (9) output rules. The user can review (10) these rules and mark some of them as erroneous if the user has sufficient evidence to do so. Then the rules are supplied (11) to the Test Script. Finally, the input space is partitioned into clusters that lead to good and bad test cases, to find methods that are specific to good performance test cases. This task is accomplished in parallel to computing rules, and it starts when the Trace Analyzer produces (12) the method and data statistics that is used to construct (13) two matrices (14). Once these matrices are constructed, ICA decomposes them

(15) into the matrices for bad and good test cases correspondingly. Finally, the Advisor (16) determines top methods that performance testers should look at (17) to debug possible performance problems.

3.2 Research Question 2: What are the main problems found by stress tests?

Performance problems share common symptoms and many performance problems described in the literature are defined by a particular set of root causes. Fig. 3.9 shows the symptoms of known performance problems [99].

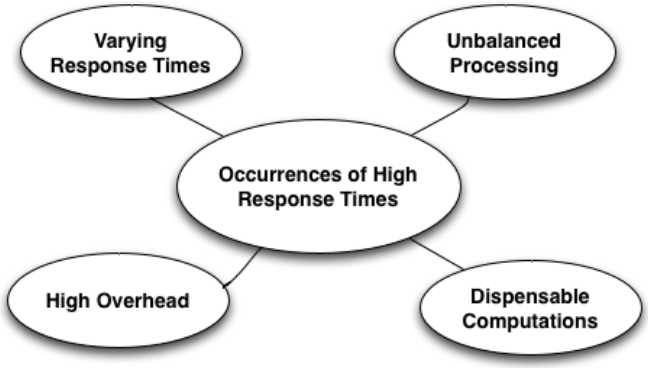


Figure 3.9: Symptoms of know performance problems [99].

There are several antipatterns that details features about common performance problems. Antipatterns are conceptually similar to patterns in that they document recurring solutions to common design problems. They are known as antipatterns because their use produces negative consequences. Performance antipatterns document common performance mistakes made in software architectures or designs. These software Performance antipatterns have four primary uses: identifying problems, focusing on the right level of abstraction, effectively communicating their causes to others, and prescribing solutions [19]. The table 3.2 present some of the most common performance antipatterns.

Blob antipattern is known by various names, including the “god” class [8] and the “blob” [2]. Blob is an antipattern whose problem is on the excessive message traffic generated by a single class or component, a particular resource does the majority of the work in a software. The Blob antipattern occurs when a single class or component either performs all of the work of an application or holds all of the application’s data. Either manifestation results in excessive message traffic that can degrade performance [24] [82].

A project containing a “god” class is usually has a single, complex controller class that is surrounded by simple classes that serve only as data containers. These classes typically contain only accessor operations (operations to get() and set() the data) and perform little or no computation of their own [82]. The Figures

Table 3.2: Performance antipatterns

antipattern	Derivations
Blob or The God Class	
Unbalanced-Processing	Concurrent processing Systems
	Piper and Filter Architectures
	Extensive Processing
Circuitous Treasure Hunt	
Empty Semi Trucks	
Tower of Babel	
One-Lane Bridge	
Excessive Dynamic Allocation	
Traffic Jam	
The Ramp	
More is Less	

3.10 and 3.11 describes an hypotetical system with a BLOB problem: The Fig. 3.10 presents a sample where the Blob class uses the features A,B,C,D,E,F and G of the hypotetical system; The Fig. 3.11 shows a static view where a complex software entity instance, i.e. Sd, is connected to other software instances, e.g. Sa, Sb and Sc, through many dependencies [93][99].

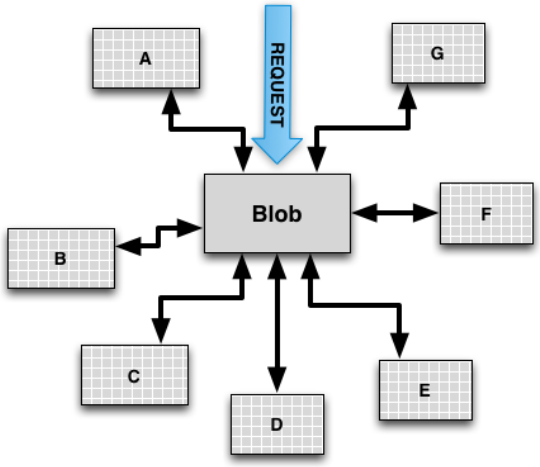


Figure 3.10: The God class[99].

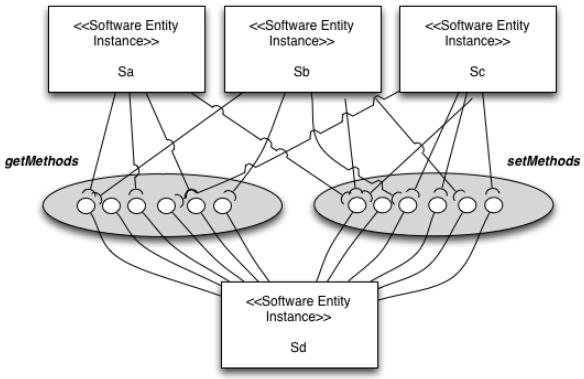


Figure 3.11: The God class[93].

Unbalanced Processing it's characterises 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. Fig. 3.21 shows a sample of the Unbalanced Processing. In The Fig. 3.21, four tasks are performed. The task D it is waiting for the task C conclusion that are submmited to a heavy processing situation.

The pipe and filter architectures and extensive processing antipattern represents a manifestation of the unbalanced processing antipattern. The pipe and filter architectures occurs when the throughput of the overall system is determined by the slowest filter. The Fig. 3.12 describes a software S with a Pipe and Filter Architectures problem: the operation opx is invoked in a service and the throughput of the service ($Th(S)) is lower than the required one. The extensive processing occurs when a process monopolizes a processor and prevents a set of other jobs to be executed until it finishes its computation. The Fig. 3.13 describes a software S with a Extensive Processing problem: the operations opx and opy are alternatively invoked in a service and the response time of the service ($RT(S)) is larger than the required one [93].

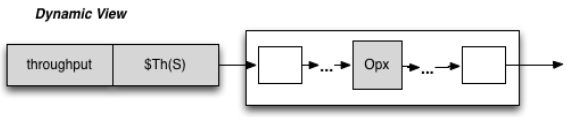


Figure 3.12: Pipe and Filter sample [93]

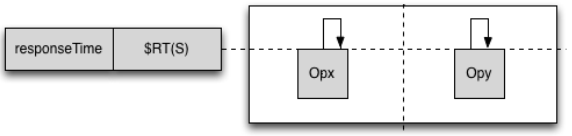


Figure 3.13: Extensive Processing sample [93].

Circuitous Treasure Hunt antipattern occurs when software retrieves data from a first componet, uses those results in a second component, retrieves data from the second component, and so on, until the last results are obtained [84] [83]. Circuitous Treasure Hunt are typical performance antipatterns that causes unnecessarily frequent database requests. The Circuitous Treasure Hunt antipattern is a result from a bad database schema or query design. A common Circuitous Treasure Hunt design creates a data dependency between single queries. For instance, a query requires the result of a previous query as input. The longer the chain of dependencies between individual queries the more the Circuitous Treasure Hunt antipattern hurts performance [100]. The Fig. 3.14 shows a software S with a Circuitous Treasure Hunt problem: the software S generates a large number of database calls by performing several queries up to the final operation [93].

Empty Semi Trucks occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both [11]. There are a special case of Empty Semi Trucks that occurs when many fields in a user interface must be retrieved from a remote system. Fig. shows a software S with a Empty Semi Trucks problem: the software instance Sa generates an excessive message traffic by sending a big amount of messages with low sizes, much lower than the network bandwidth, hence the network link might have a low utilization value [93].

The Tower of Babel antipattern most often occurs when information is translated into an exchange format, such as XML, by the sending process then parsed and translated into an internal format by the receiving process. When the translation and parsing is excessive, the system spends most of its time doing this and relatively little doing real work [83]. Fig. shows a system with a Tower of Babel problem: the software instances Sd performs many times the translation of format for communicating with other instances [93].

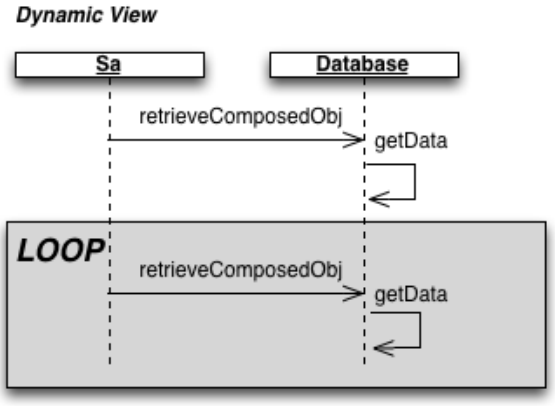


Figure 3.14: Circuitous Treasure Hunt sample [93]

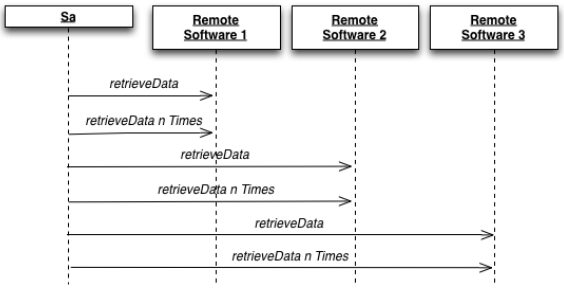


Figure 3.15: Empty Semi Trucks sample [93].

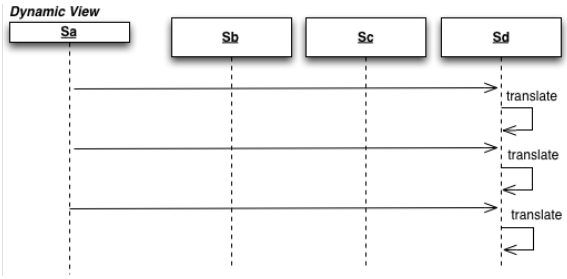


Figure 3.16: Tower of Babel sample [93]

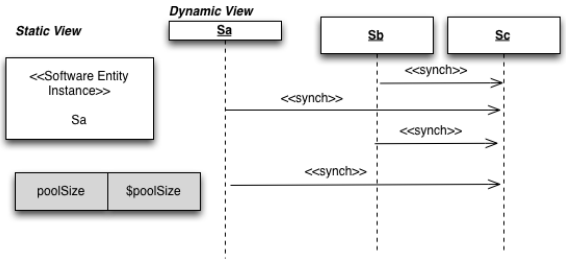


Figure 3.17: One-Lane Bridge sample [93].

One-Lane Bridge is a antipattern that occurs when one or a few processes execute concurrently using a shared resource and other processes are waiting for use the shared resource. It frequently occurs in applications that access a database. Here, a lock ensures that only one process may update the associated portion of the database at a time. This antipatterns is common when many concurrent threads or processes are waiting for the same shared resources. These can either be passive resources (like semaphores or mutexes) or active resources (like CPU or hard disk). In the first case, we have a typical One Lane Bridge whose critical resource needs to be identified. Figure 3.10 shows a system with a One-Lane Bridge problem: the software instance Sc receives an excessive number of synchronous calls in a service S and the predicted response time is higher than the required [93].

Using dynamic allocation, objects are created when they are first accessed and then destroyed when they are no longer needed. Excessive Dynamic Allocation, however, addresses frequent, unnecessary creation and destruction of objects of the same class. Dynamic allocation is expensive , an object created in memory must be allocated from the heap, and any initialization code for the object and the contained objects must be executed. When the object is no longer needed, necessary clean-up must be performed, and the reclaimed memory must be returned to the heap to avoid memory leaks [84] [83].

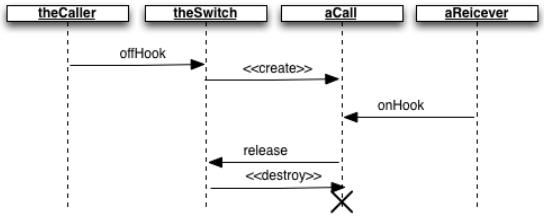


Figure 3.18: Excessive Dynamic Allocation.

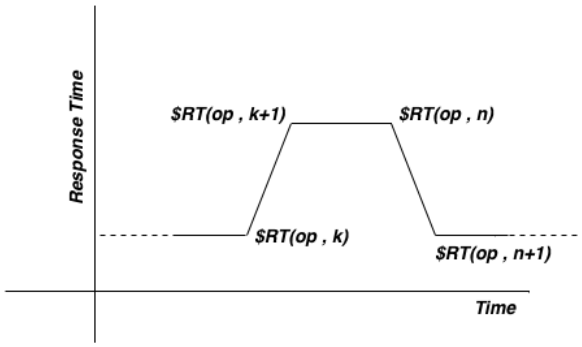


Figure 3.19: Traffic Jam Response Time [93].

The Fig. 3.18 shows a Excessive Dynamic Allocation sample. This example is drawn from a call (an offHook event), the switch creates a Call object to manage the call. When the call is completed, the Call object is destroyed. Constructing a single Call object it is not seem as excessive. A Call is a complex object that contains several other objects that must also be created. The Excessive Dynamic Allocation occurs when a switch receive hundreds of thousands of offHook events. In a case like this, the overhead for dynamically allocating call objects adds substantial delays to the time needed to complete a call.

The Traffic Jam antipattern occurs if many concurrent threads or processes are waiting for the same active resources (like CPU or hard disk). This antipatterns produces a large backlog in jobs waiting for service. The performance impact of the Traffic Jam is the transient behavior that produces wide variability in response time. Sometimes it is fine, but at other times, it is unacceptably long. Figure 3.19 describes a software with a Traffic Jam problem, the monitored response time of the operation shows a wide variability in response time which persists long [93].

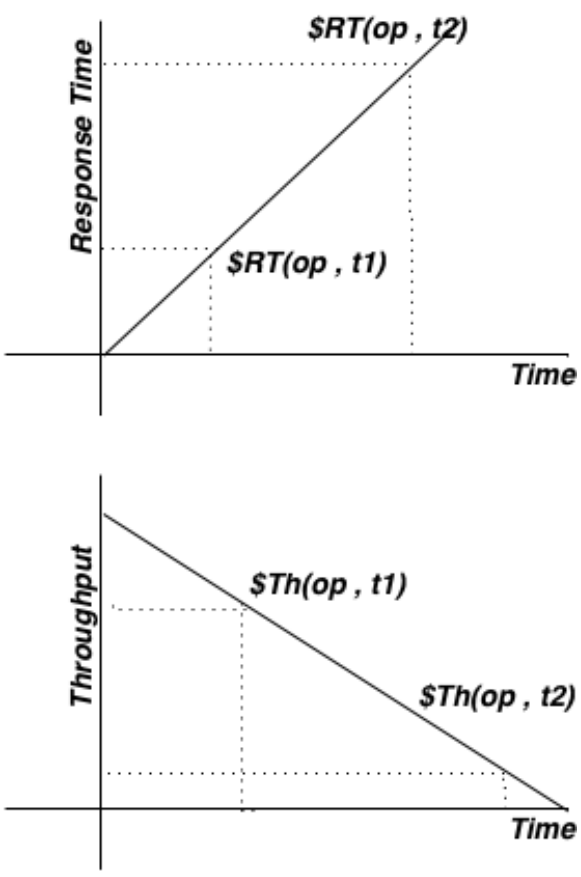


Figure 3.20: The Ramp sample [93].

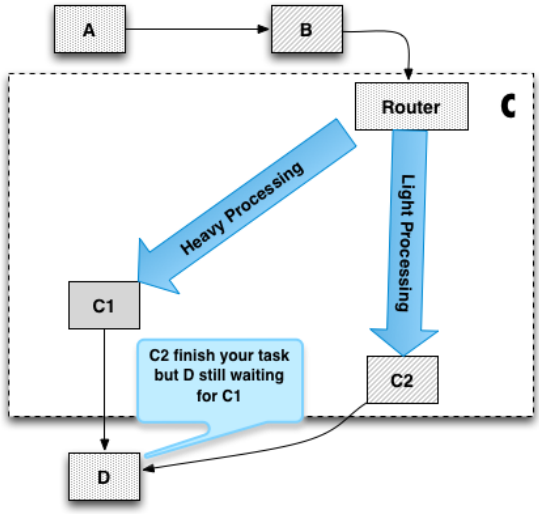


Figure 3.21: Unbalanced Processing sample [99].

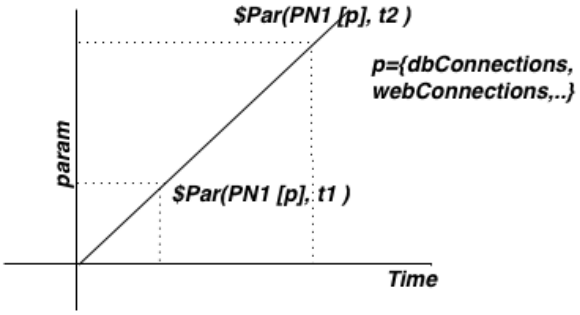


Figure 3.22: More is Less sample [93].

The Ramp it is a antipattern where the processing time increases as the system is used. The Ramp can arise in several different ways. Any situation in which the amount of processing required to satisfy a request increases over time will produce the behavior. With the Ramp antipattern, the memory consumption of the application is growing over time. The root cause is Specific Data Structures which are growing during operation or which are not properly disposed [100] [83]. Fig. 3.20 shows a system with The Ramp problem: (i) the monitored response time of the operation opx at time t1, i.e. $RT(opx, t1), is much lower than the monitored response time of the operation opx at time t2, i.e. $RT(opx, t2), with t1 < t2; (ii) the monitored throughput of the operation opx at time t1, i.e. $Th(opx, t1), is much larger than the monitored throughput of the operation opx at time t2, i.e. $Th(opx, t2), with t1 < t2.

More is less occurs when a system spends more time "thrashing" than accomplishing real work because there are too many processes relative to available resources. More is Less are presented when it is running too many programs overtime. This antipattern causes too much system paging and systems spend all their time servicing page faults rather than processing requests. In distributed systems, there are more causes.They

include: creating too many database connections and allowing too many internet connection. Fig. 3.22 describes a system with a More Is Less problem: There is a processing node PN1 and the monitored runtime parameters (e.g. database connections, etc.) at time t1, i.e. $Par(PN1[p], t1), are much larger than the same parameters at time t2, i.e. $Par(PN1[p], t2), with t1 < t2.

Chapter 4

Search-Based Stress Testing

The search for the longest execution time is regarded as a discontinuous, nonlinear, optimization problem, with the input domain of the system under test as a search space [86]. The application of SBST algorithms to stress tests involves finding the best- and worst-case execution times (B/WCET) to determine whether timing constraints are fulfilled [2].

There are two measurement units normally associated with the fitness function in a stress test: processor cycles and execution time. The processor cycle approach describes a fitness function in terms of processor cycles. The execution time approach involves executing the application under test and measuring the execution time [2] [91]. Processor cycles measurement is deterministic in the sense that it is independent of system load and results in the same execution times for the same set of input parameters. However, such a measurement is dependent on the compiler and optimizer used, therefore, the processor cycles differ from each platform. Execution time measurement is a non-deterministic approach, in which there is no guarantee of obtaining the same test inputs [2]. However, stress testing where testers have no access to the production environment should be measured by the execution time measurement [65] [2].

Table 4.1 shows a comparison between the research studies on load, performance, and stress tests presented by Afzal et al. [2]. Afzal's work was added with some of the latest research in this area ([37] [39] [27] [28] [4] [43]). The columns represent the type of tool used (prototype or functional tool), and the rows represent the metaheuristic approach used by each research study (genetic algorithm, Tabu search, simulated annealing, or a customized algorithm). The table also sorts the research studies by the type of fitness function used (execution time or processor cycles).

The studies can be grouped into two main groups: Search-Based Stress Tesing on Safety-critical systems or Search-Based Stress Testing on Industrial systems.

Table 4.1: Distribution of the research studies over the range of applied metaheuristics

	Prototypes		**Functional Tool**
	Execution Time	Processor Cycles	Execution Time
GA + SA + Tabu Search +Q-Learning			Our approach
GA + SA + Tabu Search			Gois et al. 2016 [43]
GA	Alander et al.,1998 [3] Wegener et al., 1996 and [97][53] Sullivan et al., 1998 [86] Briand et al., 2005 [18] Canfora et al., 2005 [22]	Wegener and Grochtmann, [96] Mueller et al., 1998 [66] Puschner et al. [72] Wegener et al., 2000 [98] Gro et al., 2000 [47]	Di Penta et al., 2007 [68] Garoussi, 2006 [37] Garousi, 2008 [38] Garousi, 2010 [39]
Simulated Annealing (SA)			Tracey, 1998 [90]
Constraint Programming			Di Alesio et al., 2014 [28] Di Alesio et al., 2013 [27]
GA + Constraint Programming			Di Alesio et al., 2015 [4]
Customized Algorithm		Pohlheim, 1999 [70]	

4.1 Search-Based Stress Testing on Safety-critical systems

Domains such as avionics, automotive and aerospace feature safety-critical systems, whose failure could result in catastrophic consequences. The importance of software in such systems is permanently increasing due to the need of a higher system flexibility. For this reason, software components of these systems are usually subject to safety certification. In this context, software safety certification has to take into account performance requirements specifying constraints on how the system should react to its environment, and how it should execute on its hardware platform [27].

Usually, embedded computer systems have to fulfil real-time requirements. A faultless function of the systems does not depend only on their logical correctness but also on their temporal correctness. Dynamic aspects like the duration of computations, the memory actually needed during program execution, and other synchronisation of parallel processes are of major importance for the correct function of real-time systems [53] .

The concurrent nature of embedded software makes the order of external events triggering the system tasks is often unpredictable. Such increasing software complexity renders performance analysis and testing increasingly challenging. This aspect is reflected by the fact that most existing testing approaches target

system functionality rather than performance [27]. Reactive real-time systems must react to external events within time constraints. Triggered tasks must execute within deadlines. Shousha develops a methodology for the derivation of test cases that aims at maximizing the chance of critical deadline misses [81].

The main goal of Search-Based Stress testing of Safety-critical systems is finding a combination of inputs that causes the system to delay task completion to the greastest extent possible. The followed approaches use metaheuristics to discover the worst-case execution times. Wegener et al. [97] used GAs to search for input situations that produce very long or very short execution times. The fitness function used was the execution time of an individual measured in micro seconds [97]. Alander et al. [3] performed experiments in a simulator environment to measure extreme response times of protection relay software using genetic algorithms. The fitness function used was the response time of the tested software. The results showed that GA generated more input cases with longer response times [3].

Wegener and Grochtmann performed a experimentation to compare GA with random testing. The fitness function used was the execution duration measured in processor cycles. The results showed that, with a large number of input parameters, GA obtained more extreme execution times with less or equal testing effort than random testing [53] [96]. Gro et. al. [47] presented a prediction model which can be used to predict evolutionary testability. The research confirmed that there is a relationship between the complexity of a test object and the ability of a search algorithm to produce input parameters according to B/WCET [47]. Briand et al. [18] used GA to find the sequence of arrival times of events for aperiodic tasks, which will cause the greatest delays in the execution of the target task. A prototype tool named real-time test tool (RTTT) was developed to facilitate the execution of runs of a GA. Two case studies were conducted and results illustrated that RTTT was a useful tool to stress a system under test [18].

Pohlheim and Wegener used an extension of genetic algorithms with multiple sub-populations, each using a different search strategy. The duration of execution, measured in processor cycles, was taken as the fitness function. The GA found longer execution times for all the given modules in comparison with systematic testing [70]. Garousi presented a stress test methodology aimed at increasing the chances of discovering faults related to distributed traffic in distributed systems. The technique uses as input a specified UML 2.0 model of a system, augmented with timing information. The results indicate that the technique is significantly more effective at detecting distributed traffic-related faults when compared to standard test cases based on an operational profile [37]. Alesio, Nejati and Briand describe an approach based on Constraint Programming (CP) to automate the generation of test cases that reveal, or are likely to, task deadline misses. They evaluate it through a comparison with a state-of-the-art approach based on GAs. In particular, the study compares CP and GA in five case studies for efficiency, effectiveness, and scalability. The experimental results show that, on the larger and more complex case studies, CP performs significantly better than GA. The research proposes a tool-supported, efficient and effective approach based on CP to generate stress test cases that maximize the likelihood of task deadline misses [27].

Alesio describes stress test case generation as a search problem over the space of task arrival times. The research locates the worst-case scenarios maximizing deadline misses where each scenario characterizes a

test case. The paper combines two strategies, GA and CP. The results show that, in comparison with GA and CP in isolation, GA+CP achieves nearly the same effectiveness as CP and the same efficiency and solution diversity as GA, thus combining the advantages of the two strategies. Alesio concludes that a combined GA+CP approach to stress testing is more likely to scale to large and complex systems [4].

4.2 Search-Based Stress Testing on Industrial systems

Usually, the application of Search-Based Stress Testing on non safety-critical systems deals with the generation of test cases that causes Service Level Agreement (SLA) violations.

Tracey et al. [90] used simulated annealing (SA) to test four simple programs. The results of the research presented that the use of SA was more effective with a larger parameter space. The authors highlighted the need of a detailed comparison of various optimization techniques to explore the worst-case execution time (WCET) and the best-case execution times (BCET) of the system under test [90].

Di Penta et al. [68] used GA to create test data that violated quality of service (QoS) constraints, causing SLA violations. The generated test data included combinations of inputs. The approach was applied to two case studies. The first case study was an audio processing workflow, and the second case study, a service producing charts [68].

Gois et al. proposes a hybrid metaheuristic approach using genetic algorithms, simulated annealing, and tabu search algorithms to perform stress testing. A tool named IAdapter, a JMeter plugin used for performing search-based stress tests, was developed. Two experiments were performed to validate the solution. In the first experiment, the signed-rank Wilcoxon non-parametrical procedure was used for comparing the results. The significance level adopted was 0.05. The procedure showed that there was a significant improvement in the results with the Hybrid Metaheuristic approach. In the second experiment, the whole process of stress and performance tests, which took 3 days and about 1800 executions, was carried out without the need for monitoring by a test designer. The tool automatically selected the next scenarios to be run up to the limit of the previously established six generations [43].

Chapter 5

Improving Search-Based Stress Tests

5.1 Improving Stress Search Based Testing using Hybrid Metaheuristic Approach

This section presents the Hybrid approach proposed by Gois et al. [43]. The solution proposed by Gois et al. makes it possible to create a model that evolves during the test. A plugin called iadapter was implemented for the research. IAdapter is a JMeter plugin designed to perform search-based stress tests. The plugin is available at `www.github.com/naubergois/newiadapter`.

The proposed solution model uses genetic algorithms, tabu search, and simulated annealing in two different approaches. The study initially investigated the use of these three algorithms. Subsequently, the study will focus on other population-based and single point search metaheuristics. The first approach uses the three algorithms independently, and the second approach uses the three algorithms collaboratively (hybrid metaheuristic approach).

In the first approach , the algorithms do not share their best individuals among themselves. Each algorithm evolves in a separate way (Fig. 5.1). The second approach uses the algorithms in a collaborative mode (hybrid metaheuristic). In this approach, the three algorithms share their best individuals found (Fig. 5.2). The next subsections present details about the used metaheuristic algorithms (Representation, initial population and fitness function).

5.1.1 Representation

The solution representation provides a common representation for all workloads. Each workload is composed by a linear vector with 21 positions (Figure 5.3 -❶). The first position represents an metadata with the name of an individual. The next positions represent 10 scenarios and their numbers of users (Figure 5.3 -❷). The fixed-length genome approach was chosen in reason of the ease of implementation in the JMeter tool. Each

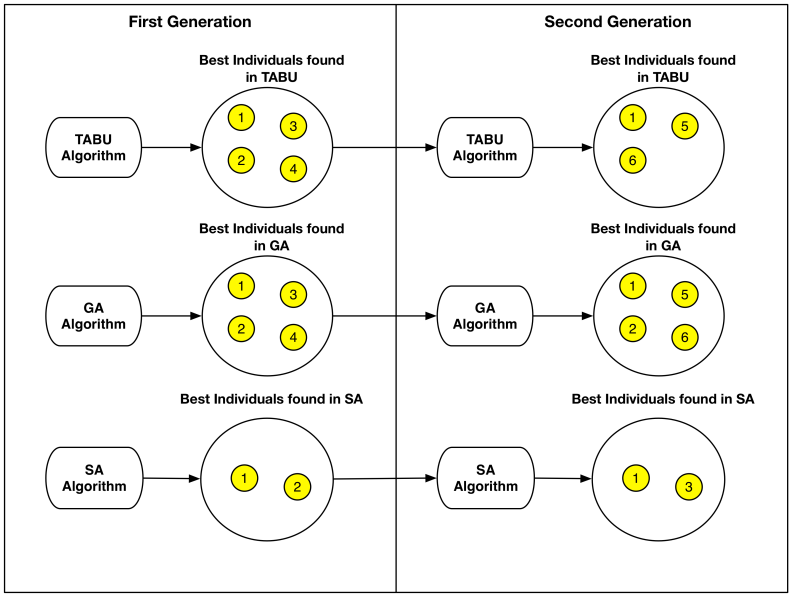


Figure 5.1: Use of the algorithms independently [43]

scenario is an atomic operation: the scenario must log into the application, run the task goal, and undo any changes performed, returning the application to its original state.

Figure. 5.3 presents the solution representation and an example using the crossover operation. In the example, solution 1 (Figure 5.3 -❸) has the Login scenario with 2 users, the Search scenario with 4 users, Include scenario with 1 user and the Delete scenario with 2 users. After the crossover operation with solution 2 (Figure 5.3 -❹), We obtain a solution with the Login scenario with 2 users, the Search scenario with 4 users, the Update scenario with 3 users and the Include scenario with 5 users (Figure 5.3 -❺). Figure. 5.3 -❻ shows the strategy used by the proposed solution to obtain the neighbors for the Tabu search and simulated annealing algorithms. The neighbors are obtained by the modification of a single position (scenario or number of users) in the vector.

5.1.2 Initial population

The strategy used by the plugin to instantiate the initial population is to generate 50% of the individuals randomly, and 50% of the initial population is distributed in three ranges of values:

- Thirty percent of the maximum allowed users in the test;
- Sixty percent of the maximum allowed users in the test; and
- Ninety percent of the maximum allowed users in the test.

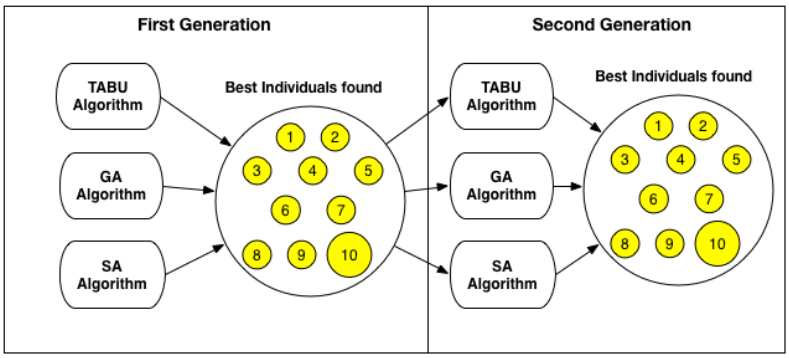


Figure 5.2: Use of the algorithms collaboratively [43]

The percentages relate to the distribution of the users in the initial test scenarios of the solution. For example, in a hypothetical test with 100 users, the solution will create initial test scenarios with 30, 60 and 90 users.

5.1.3 Objective (fitness) function

The proposed solution was designed to be used with independent testing teams in various situations, in which the teams have no direct access to the environment, where the application under test was installed. Therefore, the IAdapter plugin uses a measurement approach as the definition of the fitness function. The fitness function applied to the IAdapter solution is governed by the following equation:

$$\begin{aligned} fit = {} & numberOfUsersWeight * numberOfUsers \\ & -90percentileweight * 90percentiletime \\ & -80percentileweight * 80percentiletime \\ & -70percentileweight * 70percentiletime \\ & -maxResponseWeight * maxResponseTime \\ & -penalty \end{aligned} \tag{5.1}$$

The users and response time factors were chosen because they are common units of measurement in load test tools [78]. The proposed solution's fitness function uses a series of manually adjustable user-defined weights (90percentileweight, 80percentileweight, 70percentileweight, maxResponseWeight, and numberOfUsersWeight). These weights make it possible to customize the search plugin's functionality. A penalty is applied when the response time of an application under test runs longer that the service level. The penalty is calculated by the follow equation:

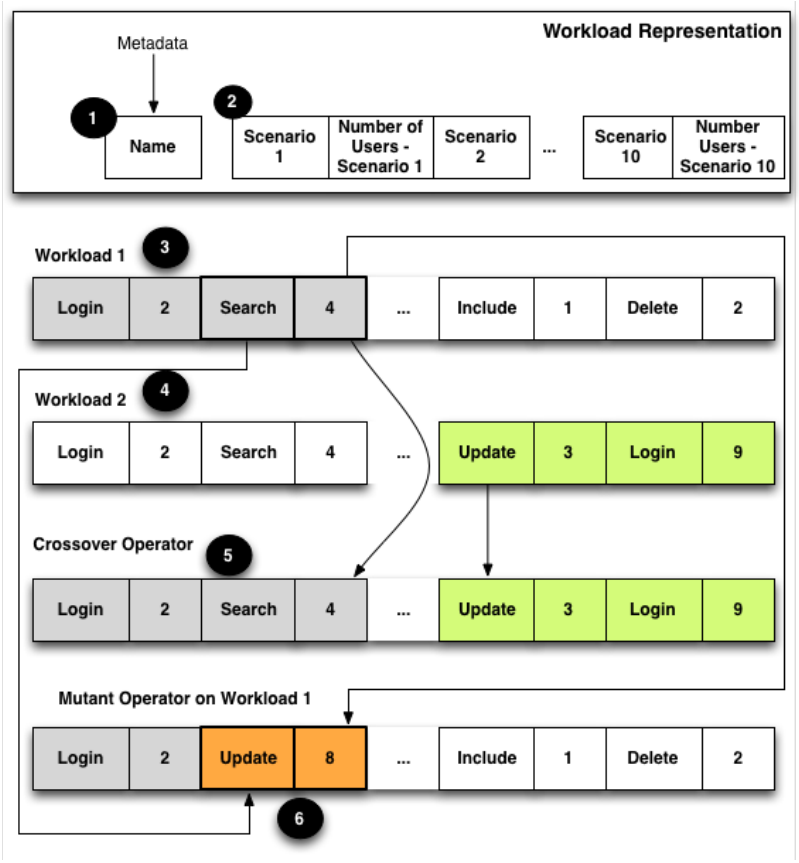


Figure 5.3: Solution representation, crossover and neighborhood operators [43]

$$penalty = 100 * \Delta$$
$$\Delta = (t_{CurrentResponseTime} - t_{MaximumResponseTimeExpected}) \tag{5.2}$$

5.2 Improving Stress Search Based Testing using Q-Learning and Hybrid Metaheuristic Approach

The goal of this research is to use a reinforcement learning technique to optimize the choice of neighboring solutions to explore, reducing the time needed to obtain the scenarios with the longest response time in the application.The research assumes that HybridQ is more expensive than Hybrid because q-learning. The reasearch has as premise that the same application under performance tests can be submitted to more than one cycle of tests execution, reducing the cost of the exploration phase of the q-learning algorithm used.

The solution, named HybridQ, uses the GA, SA and Tabu Search algorithms in a collaborative approach.

Just like most reinforcement learning problems the proposed solution works in two different phases: exploration and exploitation. The following subsections show details of the exploration and exploitation phases and the integration between metaheuristics and the Q-Learning algorithm.

5.2.1 Exploration phase

The exploration phase uses a markov model, as shown in Fig. 5.4, the proposed MDP model has three main states based on response time. A test may have a response time greater than 1.2 times the maximum response time allowed, between 0.8 and 1.2 times the maximum response time allowed or less than 0.8 times the maximum response time allowed. The values of 1.2 and 0.8 were chosen from the assumption of a tolerance margin of 20% for the application under test. This margin may be higher or lower depending on the business requirements of the application.

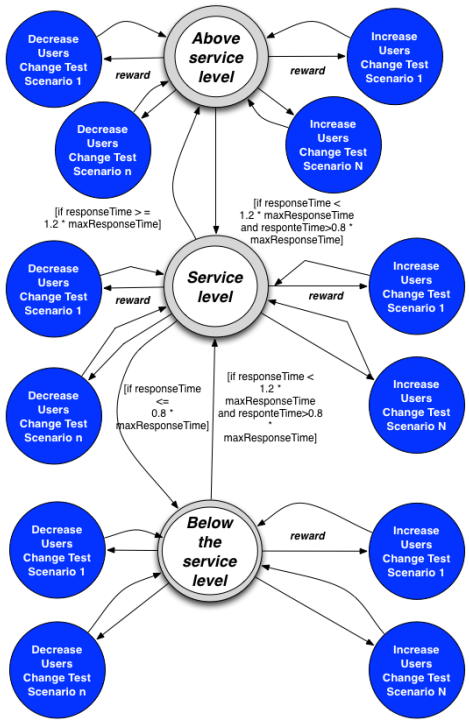


Figure 5.4: Markov Decision Process used by HybridQ

The algorithm maintains three different tables (Table 5.1), one for each state. The selection of which table to use depends on the response time of the application.

Algorithm 5 shows the main steps of exploration phase. The possible actions in MDP are the change of one of the test scenarios and an increase or decrease in the number of users. In line 1, the algorithm choose a random action (increase, decrease or maintain the number of users). In line 2, the algorithm choose one a random testScenario. In lines 3 to 7, the algorithm checks if there exists a q value for the pair (action and test

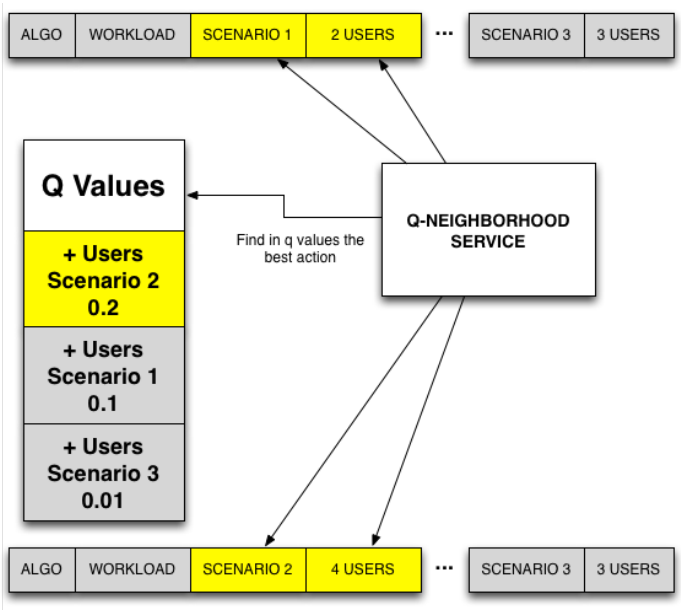


Figure 5.5: HybridQ NeighborHood Service

Algorithm 4 Exploration phase table selection

```
1: if responseTime < 0.8 * maxResponseTime  then
2:     return qTableBellowServiceLevel
3: end if
4: if responseTime >= 0.8 * maxResponseTime and responseTime <= 1.2* maxResponseTime then
5:     return qTableServiceLevel
6: end if
7: if responseTime > 1.2 * maxResponseTime then
8:     return qTableAboveServiceLevel
9: end if
```

scenario), if not exist a q value then zero value is assigned. In line 8, the algorithm checks i the new solution increases the fitness value. A solution receives a positive reward when an action increases the fitness value and a negative reward when an action reduces the fitness value. Finally, the algorithm updates the qTable with the new q value.

Unlike the traditional approach, The update of Q values for each action also occurs in the exploitation phase. The exploration phase ends when no value of Q equals zero for a state, ie, unlike the traditional approach an agent belonging to one state may be in the exploration phase while another agent may be in the explotation phase. Table 5.1 presents hypothetical Q-values for a test. In Table 5.1, it can be observed that the agents in the Service Level state are in the exploitation phase because there is no other value of Q that equals to zero.

Algorithm 5 HybridQ exploration phase

```
1:  action ← Random.createAction()
2:  testScenario ← Random.chooseTestScenario()
3:  if qTable.containsKey(action + "#" + testScenario) then
4:      qValue ← qTable.get(action + "#" + testScenario);
5:  else
6:      qValue ← 0
7:  end if
8:  if newSolution.getFitness() > oldSolution.getFitness() then
9:      qValue ← ReinforcementLearning.alpha * reward + (1 − ReinforcementLearning.alpha) * qValue
10: else
11:     qValue ← ReinforcementLearning.alpha * −reward + (1 − ReinforcementLearning.alpha) * qValue
12: end if
13: qTable.update(action + "#" + testScenario, qValue)
```

Table 5.1: Hypothetical MDP Q-values

Above Service Level	**Scenario 1**	**Scenario 2**
Increment Users	0.2	0.0
Reduce Users	0.1	0.2
Phase	Exploration	Exploration
Service Level	**Scenario 1**	**Scenario 2**
Increment Users	0.2	0.11
Reduce Users	0.1	-0.2
Phase	Explotation	Explotation
Bellow Service Level	**Scenario 1**	**Scenario 2**
Increment Users	0.0	0.2
Reduce Users	0.1	0.0
Phase	Exploration	Exploration

5.2.2 Exploitation phase

The main objective of the exploitation phase is to choose the best neighboring solution based on the Q value. The research expected that Q-Learning improve Hybrid algorithm replacing the random characteristic of the tabu search, simulated annealing and genetic algorithms operators by the direction given by the q-learning in the exploration phase. Algorithm 6 presents the main steps of exploitation phase. In first line, the algorithm gets the original genome. In lines 2 to 11, HybridQ gets the maximum, the second maximum or the third maximum q value, depending on the random value of the random variable.The algorithm chooses one of the three largest values of q. The variation of the highest values was inserted in the algorithm to escape the local optimals. In line 12, the algorithm gets the key value in Table that have the maximum q value. In line number 13, The key is separated into two parts using the # delimiter. The first part of the key is action and the second part is the test scenario. If the action equals '*up*' value, the genome is incremented in its users. If the action equals '*down*' value, the genome is incremented in its users. Finally, the test scenario is changed and the new genome is returned.

Algorithm 6 HybridQ exploitation phase

```
1: Gene[]genome ← service.getTestGenome()
2: random ← Random.nextInt(3)
3: if random==1 then
4:     qMaxValue ← qTable.getMaxValue(responseTime)
5: end if
6: if random==2 then
7:     qMaxValue ← qTable.getSecondMaxValue(responseTime)
8: end if
9: if random==3 then
10:    qMaxValue ← qTable.getThirdMaxValue(responseTime)
11: end if
12: key ← qTable.selectKey(qMaxValue)
13: String[] keySplit ← key.split('#')
14: action ← keySplit[0]
15: testScenario ← keySplit[1]
16: if action=='up' then
17:    increaseUsers(genome)
18: end if
19: if action=='down' then
20:    decreaseUsers(genome)
21: end if
22: genomePosition ← Random.nextInt(genome.length)
23: changeTestScenario(genome,testScenario,genomePosition)
```

5.2.3 Integration between metaheuristics and the Q-Learning algorithm

The Q-learning algorithm is used by Tabu Search or Simulated Annealing to obtain the neighbors and in the mutation operator of the genetic algorithm. Unlike the traditional processes of obtaining neighboring solutions such as random change and permutation, the decision to change a genome gene is made from the action that has the highest value of Q. Fig. 5.5 presents how one of the neighbors of a test is generated using Q-Learning in IAdapter. The solution uses a service called Q-Neighborhood Service to generate the neighbor from the action that has the highest value of Q.

5.3 Search-based stress testing applications using NSGA-II

The proposed solution makes it possible to create a model that performs search-based stress tests with two distinct objectives. In the solution, each workload represents an individual in the search space. The NSGA-II implementation used an adapted implementation of the jMetal framework (`http://jmetal.sourceforge.net/`).

Figure. 6.8 presents the proposed solution life cycle. Given an initial population (Figure 6.8 -❶), the

NSGA-II algorithm implementation receives a set of workloads (Figure 6.8 -❷). The NSGA-II implementation generates a new set of individuals based on crossover or/and mutant operators (Figure 6.8 -❸). JMeterEngine runs each workload (Figure 6.8 -❺) and the NSGA-II algorithm ranks and classifies each workload based on the objective functions (Figure 6.8 -❻). After all these steps the cycle begins until the maximum number of generations is reached (Figure 6.8 -❼).

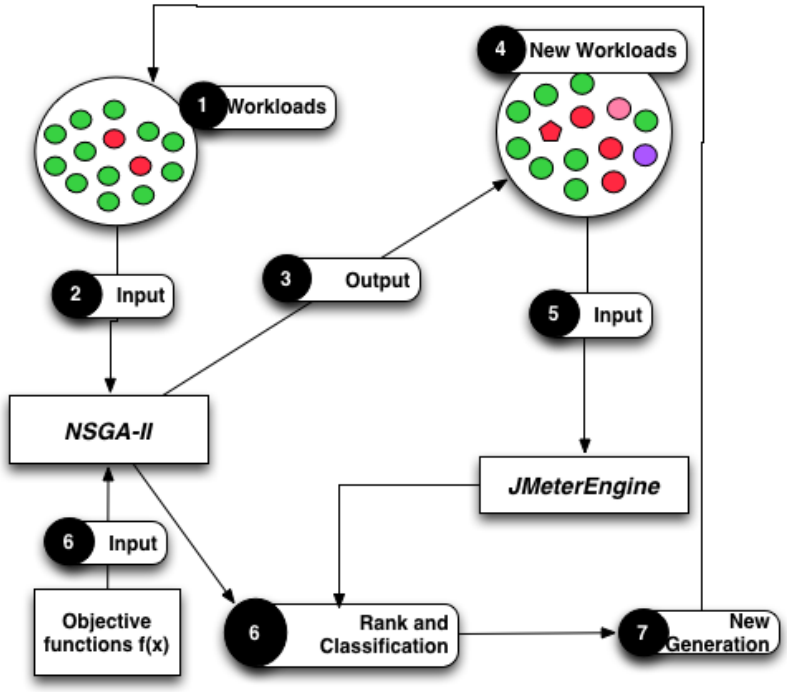


Figure 5.6: Test module life cycle.

Chapter 6

IAdapter

IAdapter is a JMeter plugin designed to perform search-based stress tests.

Fig. 6.1 presents the IAdapter Life Cycle. The main difference between IAdapter and JMeter tool is that the IAdapter provide an automated test execution where the new test scenarios are choosen by the test tool. In a test with JMeter, the tests scenarios are usually chosen by a test designer.

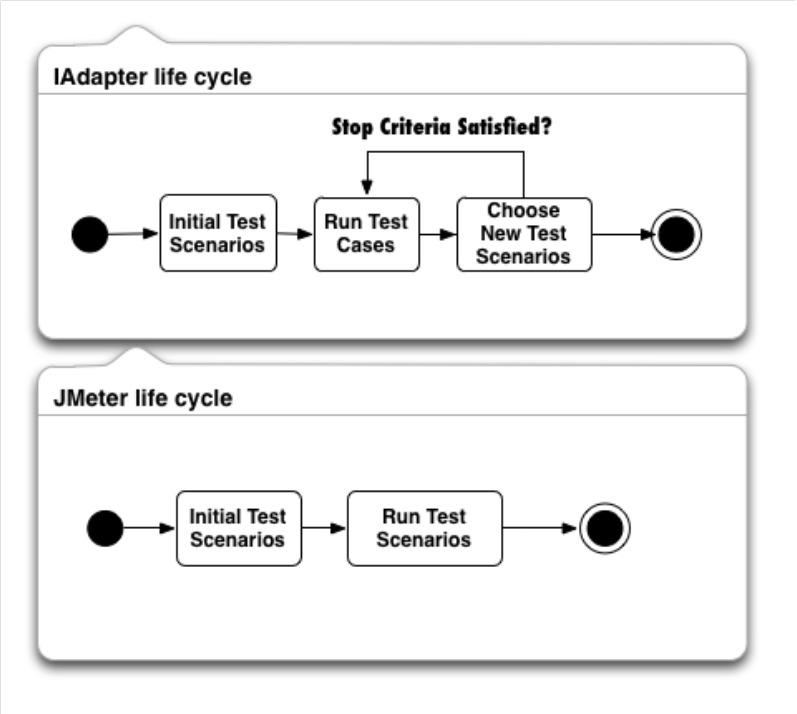


Figure 6.1: IAdapter life cycle

6.1 IAdapter Visual Components

JMeter has components organized in a hierarchical manner. The IAdapter plugin provides three main components: WorkLoadThreadGroup, WorkLoadSaver, and WorkLoadController.

WorkLoadThreadGroup is a component that creates an initial population and configures the algorithms used in the IAdapter. Fig. 6.2 presents the main screen of the WorkLoadThreadGroup component. The component has a name ❶, a set of configuration tabs ❷, a list of individuals by generation ❸, a button to generate an initial population ❹, and a button to export the results ❺. WorkLoadThreadGroup component uses the GeneticAlgorithm, TabuSearch and SimulateAnnealing classes. The WorkLoadSaver component is responsible for saving all data in the database. The operation of the component only requires its inclusion in the test script. WorkLoadController represents a scenario of the test. All actions necessary to test an application should be included in this component. All instances of the component require to be logged in the application under test and bring the application back to its original state.

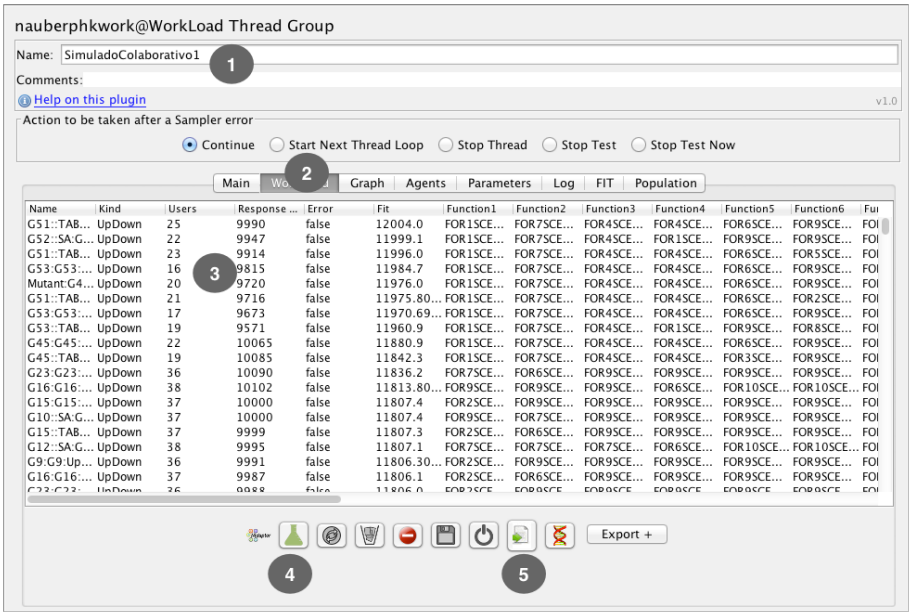


Figure 6.2: WorkLoadThreadGroup component

6.2 The IAdapter architecture

In this section, We present the IAdapter main architecture. The testbed tool proposed consists of four main modules. Figure 6.3 presents the main architecture of the solution proposed. The emulator module provides workloads to the Test module. The Test module uses a class loader to find all classes that extends AbstractAlgorithm in the classpath and run all workloads with each metaheuristic found. The Test Scenario library provides the scenario representation used by the metaheuristics and store the testbed results in a database.

The Operation services are responsible for finding neighbors of some workload provided as a parameter and perform crossover operations.

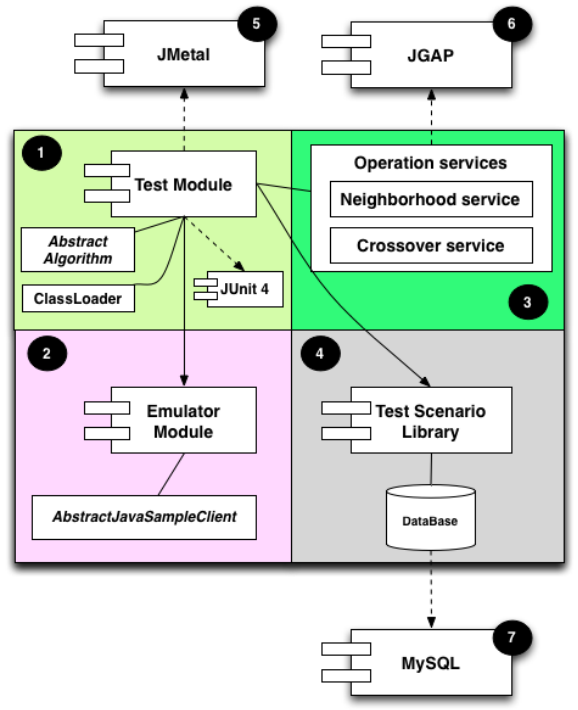


Figure 6.3: IAdapter main architecture.

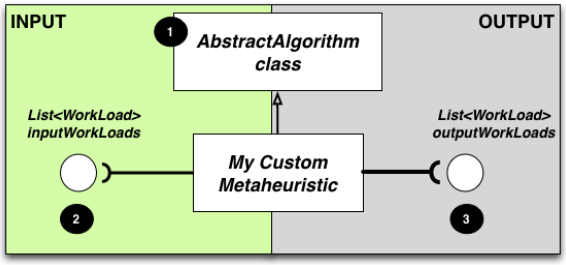


Figure 6.4: Test Module class diagram.

6.2.1 Test Module

The Test Module (Figure 6.3 -❶) is responsible for for the loading of all classes that extend AbstractAlgorithm in the classpath and perform the tests under the application. Figure 6.4 shows the class diagram for custom and provided heuristics. All heuristic classes extends the class AbstractAlgorithm. The heuristics receives as input a list of workloads (Figure 6.4 -❷) and must return a list of output workloads (the individuals selected for the next generation) (Figure 6.4 -❸). Each workload represent an individual in the search space. Figure. 6.8 presents the Test Module life cycle. Given an initial population (Figure 6.8 -❶), a metaheurist select a new set of workloads based on an objective function (Figure 6.8 -❷). The choosen metaheurist generate a new set of individuals based on crossover or neighborhood operators (Figure 6.8 -❸). JMeterEngine run each workload (Figure 6.8 -❺) and the choosen metaheuristic obtain a fitness value for each workload based on some objective function (Figure 6.8 -❻). Each Metaheuristic could define your own objective function. After all these steps the cycle begins until the maximum number of generations it is reached (Figure 6.8 -❼).

The WorkLoadThreadGroup class is responsible for start all threads that go simulate the users in the jmeter engine. Fig presents the WorkLoadThreadGroup life cycle. First, an instance of the class waits for the

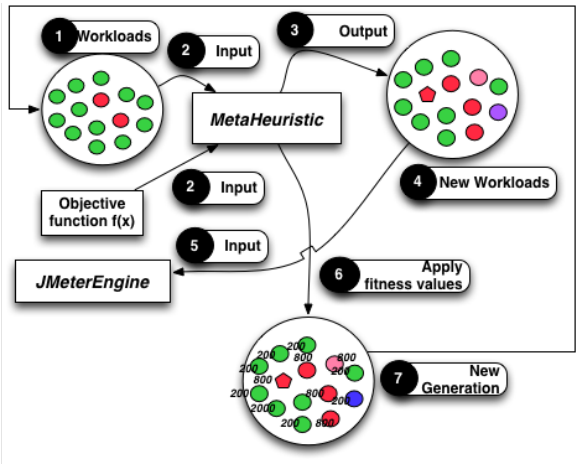


Figure 6.5: Test module life cycle.

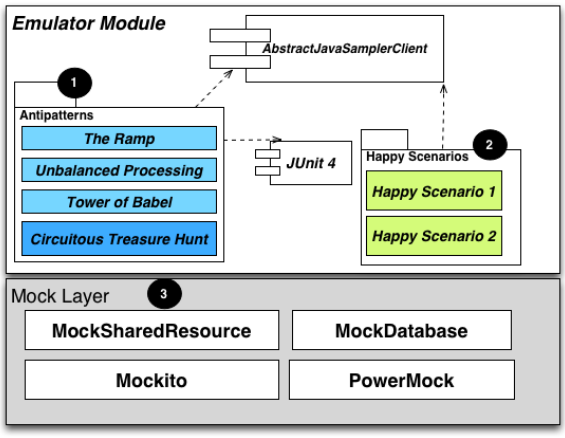


Figure 6.6: Emulator module

user to start the test (). Once the test is started, the start method is called and the instance goes to Running state.

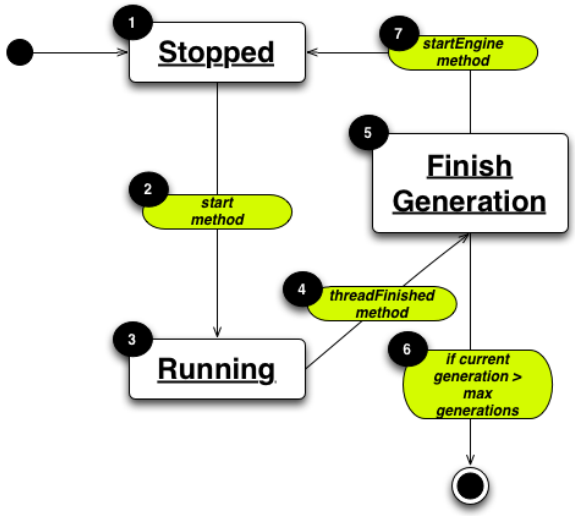


Figure 6.7: Test module life cycle.

Figure presents the sequence diagram to the start method. The start method is responsable for load the multi-objective weights, synchronize all threads and get the new workloads from database. The multi-objective weights are stored in a static variable named

6.2.2 Emulator Module

The Emulator Module is responsible for implementing and providing successful scenarios and the most common performance antipatterns (Figure 6.3 -❷). All classes must extend the AbstractJavaSamplerClient class or use JUnit 4. The AbstractJavaSamplerClient class allows the creation of a JMeter Java Request. Figure 6.11 presents the main features of the emulator module. The module implements 2 happy scenarios (Figure 6.11 -❷) and 4 antipatterns test scenarios (Figure 6.11 -❶), in its first version. The Mock Layer provides emulated databases and components for the test scenarios. The Mock Layer use the Mockito and PowerMocks frameworks (Figure 6.11 -❸).

6.2.3 Test Scenario Library

This modules provides a common representation for all workloads. The representation of each individual is encapsulated in the WorkLoad class (Fig **??**).

6.2.4 Operation services

The services are responsible for performing some operations performed by metaheuristics.

6.2.5 External dependencies

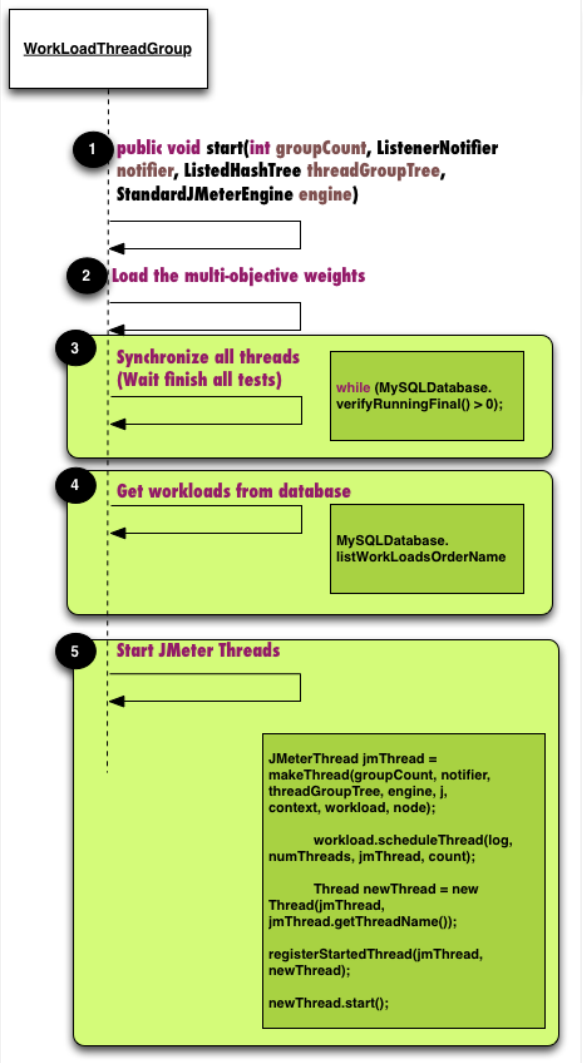


Figure 6.8: Test module life cycle.

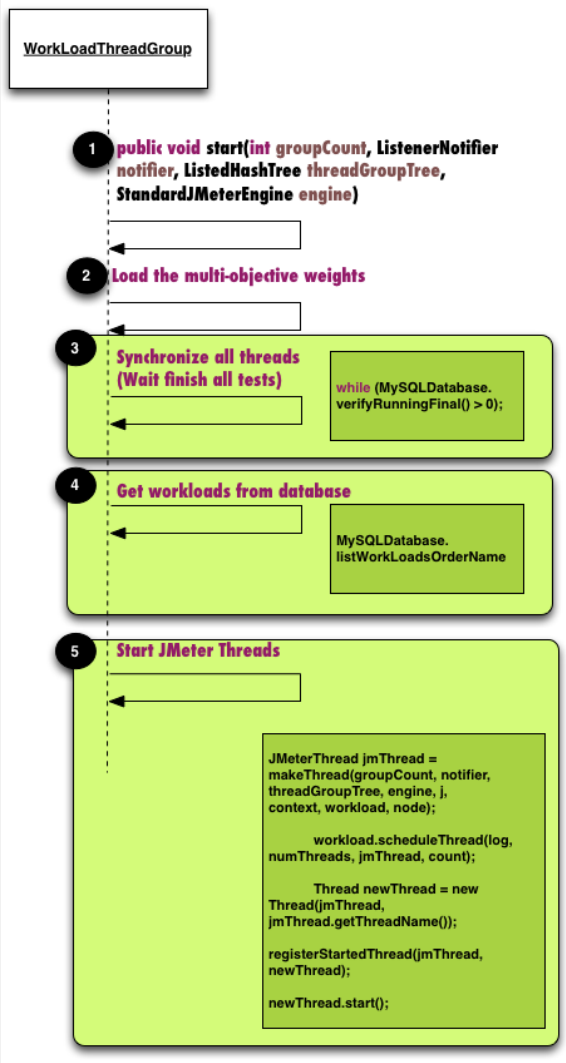


Figure 6.9: Emulator module

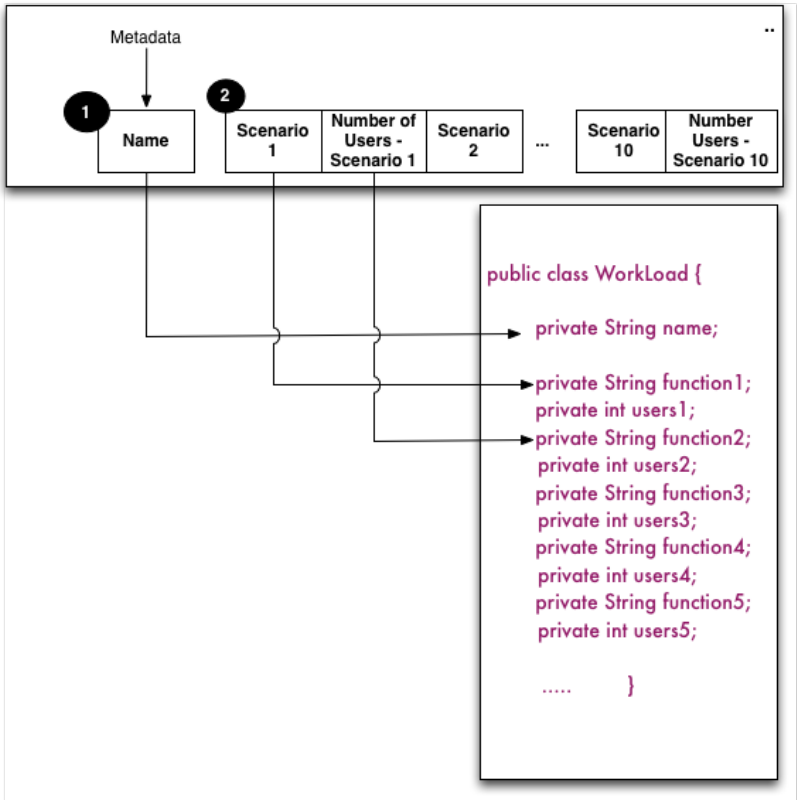


Figure 6.10: WorkLoad class

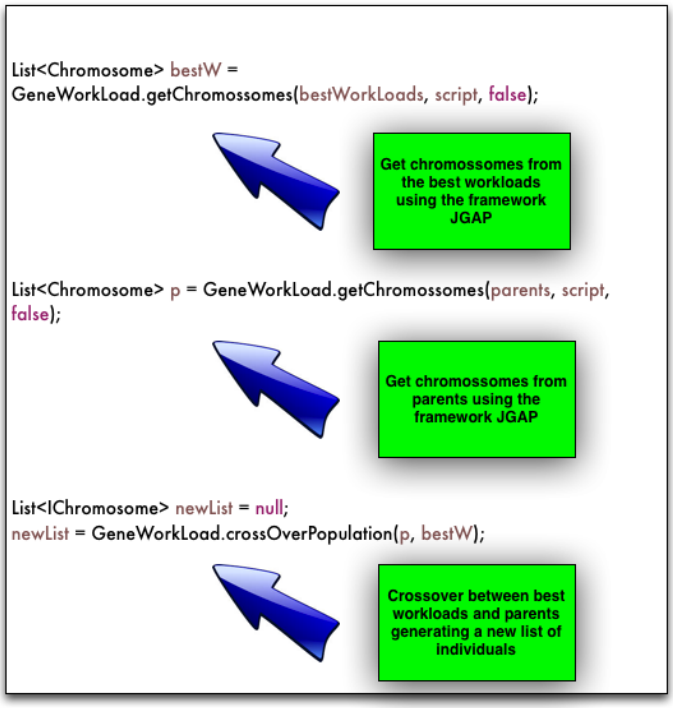


Figure 6.11: WorkLoad class

Chapter 7

Experiments

7.1 Experiments with Hybrid Algorithm

This section presents two experiments. The first one was performed on an emulated component, and the second one was performed using an installed Moodle application. The experiments used the following fitness function:

$$\begin{aligned} fit = {} & 0.9 * 90percentiletime \\ & +0.1 * 80percentiletime \\ & +0.1 * 70percentiletime+ \\ & 0.1 * maxResponseTime+ \\ & 0.2 * numberOfUsers - penalty \end{aligned} \tag{7.1}$$

This fitness function is the same function represented in the section VII with the manually adjustable user-defined weights filled out. This fitness function intended to find individuals with the highest percentile of 90%, followed by individuals with a higher percentile time of 80% and 70%, maximum response time, and number of users.

The first experiment ran for 27 generations, and the second experiment performed 6 generations, with 300 executions by generation (100 times for each algorithm), generating 300 new individuals. The experiments used an initial population of 100 individuals. The genetic algorithm used the top 10 individuals from each generation in the crossover operation. The Tabu list was configured with the size of 10 individuals and expired every 2 generations. The mutation operation was applied to 10% of the population on each generation.

Listing 7.1 SimulateConcurrentAccess class

```
1:  public class SimulateConcurrentAccess {
2:    @Test
3:    public void firstScenario() {
4:      synchronized (StaticClass.class) {
5:        for (int i = 0; i <= 1000; i++) {
6:          StaticClass.x += i;
7:        }
8:        StaticClass.x = 0;
9:      }
10:   }
11:
12:     @Test
13:   public void secondScenario() {
14:     synchronized (StaticClass.class) {
15:       for (int i = 0; i <= 2000; i++) {
16:         StaticClass.x += i;
17:       }
18:       StaticClass.x = 0;
19:     }
20:   }
```

7.1.1 First Experiment: Emulated Class Test

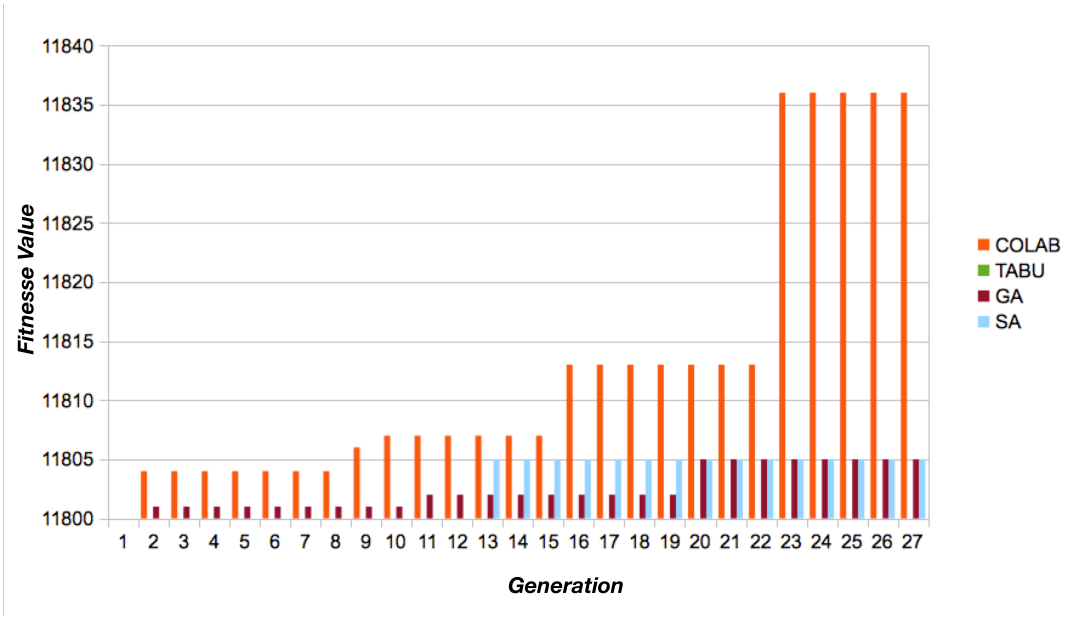
The first experiment aimed to perform performance, load, and stress testing on a simulated component. The purpose of using a simulated component was to be able to perform a greater number of generations in a shorter time available and eliminate variables such as the use of databases and application servers. The first experiment used a test class named SimulateConcurrentAccess. This class has a static variable named *x* and a set of methods that use the variable in a synchronized context (Listing 7.1). The experiment was executed using the JMeter Java Request Sampler Component with IAdapter.

Fig.7.1 presents the best results in 27 generations applied in the first experiment. The figure shows the results obtained with the algorithms with and without collaboration. The *x* axis represents the generation number, and the *y* axis represents the best fitness value obtained until the current generation. A higher value in the figure means that the scenario has a greater response time by the application under test. The results of the experiment showed that the use of cooperation between the three algorithms resulted in finding the individuals with better fitness values.

Table 7.1 presents the results obtained by the hybrid metaheuristic (HM) approach, genetic algorithm (GA), simulated annealing (SA), and Tabu search (TS) from 27 generations in the first experiment. The values are the maximum fitness value obtained by each algorithm.

The signed-rank Wilcoxon non-parametrical procedure was used for comparing the results with Z-value and W-value. The significant level adopted was 0.05. The Z-value obtained was -2.2736 and the p-value was 0.0232. The W-value obtained was 78. The critical value of W for N = 25 at p<= 0.05 was 89.The result was significant at p<= 0.05. The procedure showed that there was a significant improvement in the results with the collaborative approach.

Figure 7.1: Best results obtained in 27 generations



7.1.2 Second Experiment: Moodle Application Test

The second experiment used a Moodle application installed in a machine with 500 GB of hard disk space and 8 GB of memory. The study used six application scenarios:

- PostDeleteMessage: This scenario posts and deletes messages in the Moodle application.
- MyHome: This scenario accesses the homepage of the user's application.
- Login: This scenario is responsible for user authentication by the application.
- Notifications: This scenario involves entering the notification page of each user.
- Start Page: This scenario shows the initial start page of the application.
- Badge: This scenario involves entering the badge page.

The maximum tolerated response time in the test was 30 seconds. Any individuals who obtained a time longer than the stipulated maximum time suffered penalties. The whole process of stress and performance tests, which took 3 days and about 1800 executions, was carried out without the need for monitoring by a test designer. The tool automatically selected the next scenarios to be run up to the limit of six generations previously established.

Table 7.2 presents the maximum fitness value obtained by the hybrid metaheuristic (HM) approach, genetic algorithm (GA), simulated annealing (SA), and Tabu search (TS) in each generation.

Table 7.1: Maximum value of the fitness function by algorithm

GEN	HM	TS	GA	SA
1	11238	11238	11238	11238
2	11804	11596	11801	10677
3	11787	8932	8411	10869
4	11723	9753	9611	10760
5	8164	9780	10738	4794
6	11802	9781	11086	6120
7	9985	5782	11272	11798
8	11803	11749	10084	11309
9	11806	7284	11633	10766
10	11807	9386	11717	4557
11	11802	9653	11802	11151
12	11807	10594	11793	9434
13	11802	10848	10382	11805
14	11801	11551	7219	10237
15	11807	1701	7189	9338
16	11813	6203	11758	5321
17	11805	10720	10805	11748
18	9600	6371	11698	7818
19	11733	8160	11648	11509
20	9589	9428	11805	4813
21	11800	9463	11798	10801
22	11805	11799	11804	6029
23	11836	11655	11800	3579
24	11805	11512	11803	5761
25	11804	11573	11802	9680
26	11800	11575	11403	9388
27	11805	10691	11745	9465

The small number of samples of the experiment is insufficient to give a statistical significance to the results of the Wilcoxon procedure. However, it is noted that, in four of six generations, the collaborative approach presented the best values. The experiment succeeded in finding 29 individuals whose maximum time expected by the application was obtained. Table 7.3 shows an example of the six individuals with the highest fitness values in the second experiment. The table shows the fitness value (Fit); the name of the scenario (Scenario); the number of users (Users); and the percentiles of 90%, 80%, and 70% (90per, 80per and 70per) in seconds.

Table 7.4 presents the percentage of genes in all test scenarios by generation with and without collaboration. Most of the genes converged to the MyHome feature, which had the highest application response time.

Table 7.2: Results obtained from the second experiment

GEN	HM	TS	GA	SA
1	32242	32242	32242	32242
2	34599	32443	26290	35635
3	35800	34896	34584	34248
4	35782	34912	32689	25753
5	35611	31833	34631	8366
6	35362	35041	33397	9706

Table 7.3: Example of individuals obtained in the second experiment

Id	Fit	Scenario	Users	90per	80per	70per
1	35800	MyHome	31	30	29	10
		Badges	4			
2	35795	MyHome	30	30	29	10
		Notifications	2			
		Badges	2			
3	35782	MyHome	32	30	29	10
		Badges	3			
4	35773	MyHome	22	30	29	10
		Notifications	6			
		Badges	9			
5	35771	MyHome	28	30	29	9
		Badges	6			
6	35683	MyHome	27	30	29	8
		Badges	10			

7.2 Experiment with HybridQ Algorithm

We conducted one experiment in order to verify the effectiveness of the HybridQ. The iterated racing procedure (irace) was applied as an automatic algorithm conguration tool for tunning metaheuristics parameters. Iterated racing is a generalization of the iterated F-race procedure to automatize the arduous task of configuring the parameters of an optimization algorithm [61]. The best parameters obtaind from irace was a population size of 5 individuals, a crossover value of 0.7551, a mutation value of 0.7947, an elitism value of 0.5356 and the maximum number of iterations of 16. The experiment ran for 16 generations in an docker evinronment on a server with 16 Gb of memory and 500 Gb hard disk. The experiment used an initial population of 5 individuals by metaheuristics. The genetic algorithm used the top 4 individuals from each generation in the crossover operation. The Tabu list was configured with the size of 10 individuals and expired every 2 generations. The mutation operation was applied to 79% of the population on each generation. The experiments use tabu search, genetic algorithms, simulated annealing, the hybrid metaheuristic approach proposed by Gois et al. [43] and the HybridQ approach.

The objective function applied is intended to maximize the response time of the scenarios being tested. In these experiments, better fitness values coincide with finding scenarios with higher values of response

Table 7.4: Percentage of genes in each scenario by generation

Gen/ Scenarios	Non collaboration approach						
	Initial	1	2	3	4	5	6
Badges	20	18	16	24	15	16	17
MyHome	**15**	**59**	**55**	**48**	**53**	**50**	**52**
StartPage	15	10	12	11	20	18	19
Notifications	25	5	11	10	9	10	9
Post	8	3	1	3	1	2	1
Login	17	5	5	4	2	4	2
	Collaboration approach						
Badges	20	29	16	25	9	16	9
MyHome	**15**	**29**	**69**	**49**	**74**	**66**	**76**
StartPage	15	22	10	21	10	10	8
Nofications	25	10	1	1	2	1	3
Post	8	2	1	1	1	2	1
Login	17	8	3	3	4	5	3

time. A penalty is applied when the response time is greater than the maximum response time expected. The experiment used the following fitness (goal) function:

$$\begin{aligned} fitness = {} & 20 * 90 percentiletime \\ & 20 * 80 percentiletime \\ & 20 * 70 percentiletime \\ & 20 * maxResponseTime \\ & -penalty \end{aligned} \tag{7.2}$$

For the experiment an objective function with a single factor was chosen, since users and response time are conflicting factors. All tests in the experiment were conducted without the need of a tester, automating the process of executing and designing performance test scenarios.

7.2.1 Experiment Research Questions

The following research question is addressed:

- Is Q-learning technique improve the choice of neighboring solutions, improving the number of requests and the time needed to find scenarios with the longest response time in the application under test?

7.2.2 Variables

The independent variable is the algorithms used in each experiment. The dependent variables are: the optimal solution found by each algorithm, the number of requests to find optimal solution and the time of execution needed by each algorithm.

7.2.3 Hypotheses

- With regard to the optimal solution found by each algorithm:
  - $H1_0$ (null hypothesis) : The HybridQ does not find better solution than the other metaheuristic approaches.
  - $H1_1$: The HybridQ finds better solution than the one discovered by other metaheuristic approaches.
- With regard to the time consumed to find the optimal solution of each algorithm:
  - $H2_0$ (null hypothesis) : The HybridQ algorithm realizes more requests than the other algorithms in the experiments performed.
  - $H2_1$: The HybridQ algorithm does not realize more requests than the other algorithms in the executed experiments.
- With regard to the number of requests needed to find the optimal solution of each algorithm:
  - $H3_0$ (null hypothesis) : The HybridQ algorithm needs more time to find the optimal solution than the other algorithms in the experiments performed.
  - $H3_1$: The HybridQ algorithm does not need more time to find the optimal solution than the other algorithms in the experiments performed.

7.2.4 Experiment phases

The experiment was conducted in two phases. The first phase verified the number of requisitions and time required for the HybridQ exploration phase. The second phase ran the stress test using GA, Tabu Search, Simulated Annealing, Hybrid and HybridQ algorithms simultaneously.

7.2.5 OpenCart Experiment

The experiment was conducted to test the use of the HybridQ algorithm in a real implemented application. The chosen application was the OpenCart application , available at `opencart.com`. OpenCart is free open source ecommerce platform for online merchants. OpenCart works with PHP 5 and MySQL. The maximum tolerated response time in the test was 5 seconds. The whole process of stress and performance tests, which run for 2 days and with about 1.500 executions, was carried out without the need for monitoring by a test designer. The tool automatically selected the next scenarios to be run up to the limit of eleven generations previously established.

The experiments use the follow application features:

- Main page: The main page of the application.

Table 7.5: Q values for response times bellow than service level

Action	Feature	Q- value	State	Feature	Q- value
up	Main Page	-0.0405763	**up**	**Add to Cart**	**0.0390237**
down	Main Page	0.00079202	down	Add to Cart	-0.00079202
same	Main Page	-0.0398	same	Add to Cart	-0.0398
up	Search Page	-0.00079202	up	View Cart	-0.0398
down	Search Page	-0.0398	down	View Cart	-0.0398
same	Search Page	-0.0398	same	View Cart	-0.0398
up	Product Detail	-0.00079202	up	Remove Item	-0.0398
down	Product Detail	-0.00079202	down	Remove Item	-0.0398
same	Product Detail	-0.0398	same	Remove Item	-0.0398

- Search item: The application searches a product.
- Product detail: The application shows detais about one item product.
- Add to Cart: The application adds a product to shopping cart.
- View Cart:The application displays the shopping cart.
- Remove Item: The application remove item from shopping cart.

Q-Learning Training Phase

The application was submitted to 1 hour of training with the Q-learning algorithm using all test scenarios and was obtained the Table 7.5 with the values of q for response times bellow than service level. The action and state with best q-value is increment the number of users ('up') in Add to Cart feature. The learning phase required 1.431 requisitions for application under test.

Results

Figure 7.2 presents the number of requests by maximum fitness value. HybridQ algorithm obtains the maximum value of fitness: 364860 ($H1_1$ hypothesis). HybridQ obtained a solution with greater fitness value, but it needs a much greater number of requests than the other algorithms, not contemplating the hypothesis $H2_1$. GA is the algorithm that obtain the best fitness value with minor number of requests ($H1_0$ hypothesis). All algorithms consume the same time of test (6 hours). The scenario with more fitness value has 4,8 seconds of response time and 38 users:

- 25 users on search page;
- 10 users on Add to Cart feature;
- 2 users removing items from cart;

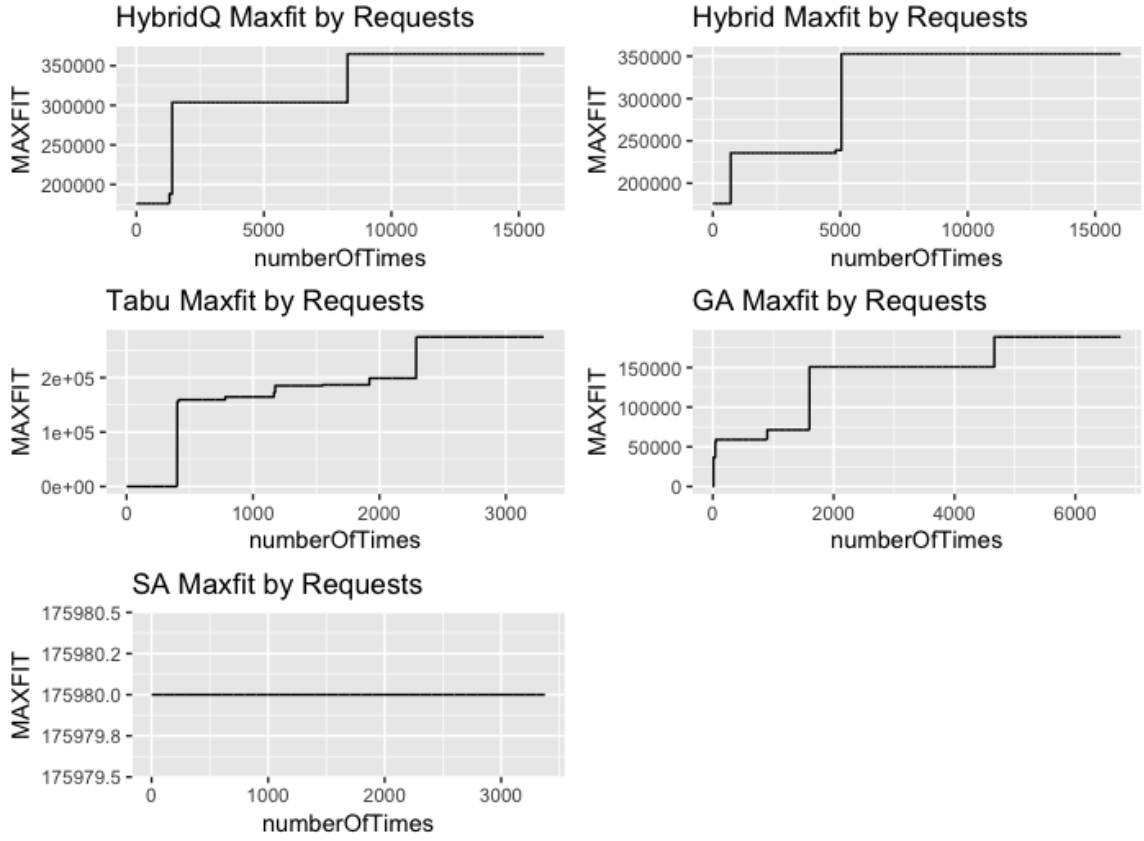


Figure 7.2: Maximum fitness value by number of requests

- 1 users on Main Page.

The t-Test and Wilcoxon Rank Sum Test was applied using the R language. The test results show that HybridQ and HybriD algorithms is superior than GA, Tabu Search and Simulated Annealing with $p < 0.02$. t-Test shows that the mean of HybridQ fitness value is superior than Hybrid.

```
 1:         Welch Two Sample t-test
 2:
 3: data:  b\$MAXFIT and c\$MAXFIT
 4: t = 13.829, df = 31678, p-value < 2.2e-16
 5: alternative hypothesis: true difference in means is not equal to 0
 6: 95 percent confidence interval:
 7:  7506.846 9986.226
 8: sample estimates:
 9: mean of x mean of y
10:  322007.5  313260.9
```

7.2.6 Threats to validity

In this work, we just evaluate the use of single objective algorithm. However, several multiobjective algorithms could be applied. The experiments are performed with the configuration obtained by the irace algorithm, however new experiments are required to verify the sensitivity of the results. It is necessary to compare the current approach with the constraint programming approaches presented in the state of art.

7.3 Experiment with multi-object NSGA-II algorithm

In this section, we present the experimental results, in which we carried out to verify the multi-objective NSGA-II implementation. The experiment was conducted to validate the use of NSGA-II multiobjective algorithm with a real implemented application. The chosen application was the JPetStore, available at `https://hub.docker.com/r/pocking/jpetstore/`. The maximum tolerated response time in the test was 500 seconds. The whole process of stress tests, which run for 3 days and 492 executions, was carried out without the need for monitoring by a test designer. The tool automatically selected the next scenarios to be run up to the limit of 123 previously established generations by the algorithm. The experiments use the follow application features:

- Enter in the Catalog: the application presents the catalog of pets.
- Fish: The application shows all the fish items in stock.
- Register: a new user is registered into the system.
- Dogs: The application shows all the dog supplies in stock.
- Shopping Cart: the application displays the shopping cart.
- Add or Remove in Shopping Cart: the application adds and removes items from the shopping cart.

The experiments used an initial population of 17 individuals by metaheuristic. The genetic algorithm used the top 10 individuals from each generation in the crossover operation The mutation operation was applied to 10% of the population in each generation. The objective function applied is intended to minimize the number of users and maximize the response time of the scenarios being tested. A penalty is applied when an application under test takes a longer time to respond than the expected maximum response time. The experiment used the following objective equations:

$$objectivefunction1 = -3 * numberOfUsers - penalty \tag{7.3}$$

$$objectivefunction2 = responsetime - penalty \tag{7.4}$$

The first objective function seeks to find workloads with fewer users. The second objective function seeks to find workloads with longer response times. The penalty is calculated by the follow equation:

$$\begin{aligned} penalty &= 100 * \Delta \\ \Delta &= (t_{CurrentResponseTime} - t_{MaximumResponseTimeExpected}) \end{aligned} \tag{7.5}$$

7.3.1 Experiment Research Questions

The following research question is addressed:

- Does the NSGA-II algorithm find relevant workload scenarios according to the two test objectives?

7.3.2 Variables

The independent variable is the NSGA-II algorithm used in the test. The dependent variables are: the optimal workload scenario found by the algorithm.

7.3.3 Hypotheses

- With regard to multi-objective algorithms applied in the experiment:
  - H_0 (A null hypothesis) : The NSGA-II didn't find workloads that meet the two objective functions used in the experiment.
  - H_1 : The NSGA-II algorithm found workloads that meet the two objective functions used in the experiment.

7.3.4 Results

Fig. 7.3 and Table 7.6 present the results obtained in the experiment. The experiment found 9 optimal workloads (Pareto Frontier) that present a lower number of users with high response times. Workload number 1 with a single user accessing the dog scenario provided a response time of 245 seconds. Workload number 2 with a single user accessing the dog scenario, 7 users accessing the Enter Catalog feature and 3 users in register functionality, provided a response time of 400 seconds.

7.3.5 Threats to validity

In this work, we just evaluate the use of one multi-objective algorithm. However, several multi-objective algorithms could be applied. There is still a reasonable distance between the Pareto frontier and the data obtained for the second objective, and more experiments are needed to validate the results.

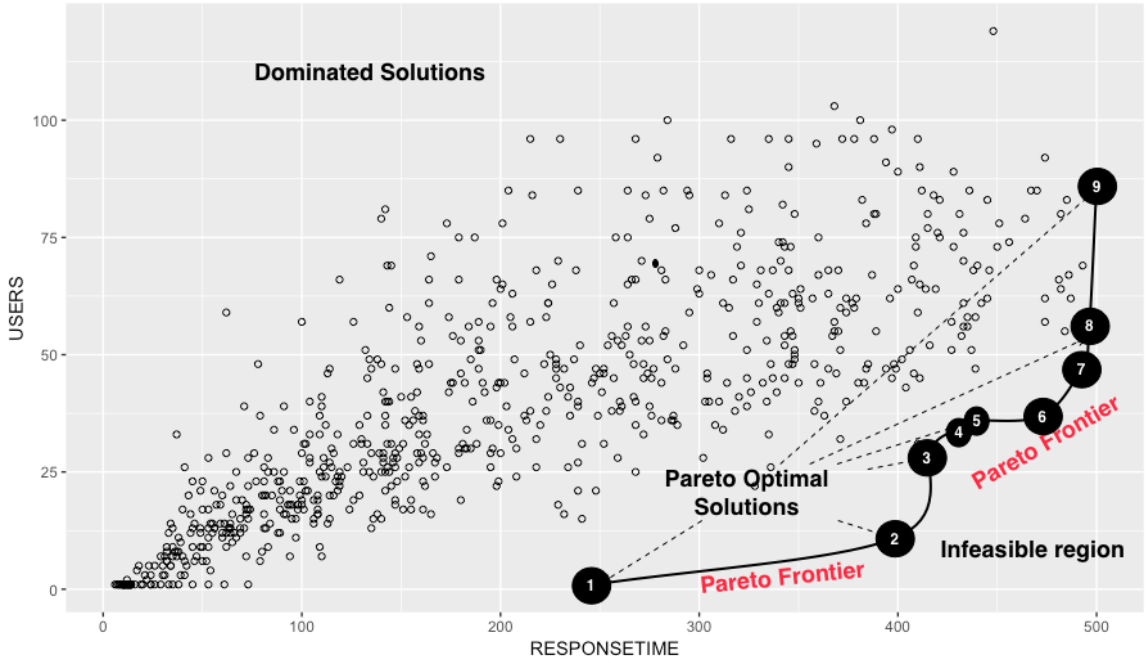


Figure 7.3: Experiment Pareto Frontier

Table 7.6: Pareto Frontier workload results

N.	OBJ. 1	OBJ. 2	Dogs Users	Enter Catalog Users	Fish Users	Register Users	Add Rem. Cart	Cart Users
❶	-3	245	1					
❷	-33	400	1	7		3		
❸	-87	416	5	15	4	5		
❹	-102	434	16	17				1
❺	-105	436	15	4	8	3	5	
❻	-111	472	7	13	7	3	7	
❼	-141	493	7	11	11	7	7	4
❽	-112	496	6		12	8	19	9
❾	-255	499		54	12		7	12

7.3.6 Conclusion

The experiment verified the use of a multi-objective algorithm in a search-based stress testing problem. A tool named IAdapter, a JMeter plugin for performing search-based load tests, was extended. One experiment was conducted to validate the proposed approach. The experiment uses the NSGA-II algorithm to discover application scenarios where there is a high response time for a small number of users. The whole process of stress tests, which ran for 3 days and 492 executions, was carried out without the need for monitoring by

a test designer. The experiment found 9 optimal workloads that present a lower number of users with high response times. The results of the experiment can help in the decision making of service levels that need to be defined for the application. For future work we intend to research using other algorithms such as SPEA2, PAES, PESA-II.

Chapter 8

Conclusion

In this thesis we dealt with the use of hybrid metaheuristics and Q-Learning in Stress Testing. This thesis presented an hybrid and an hybrid with Q-Learning metaheuristic approaches that combines genetic algorithms, simulated annealing, and tabu search algorithms in stress tests. A tool named IAdapter (github.com/naubergois/newiadapter), a JMeter plugin for performing search-based load tests, was developed. Six experiments were conducted to validate the proposed approach. The first experiment was performed on an emulated component. The second and third experiments are conducted in an testbed developed appplication. The fourth experiment was performed using an installed Moodle application. The fifth and sixth experiments are performned using an installed JPetStore application.

IAdapter Testbed is an open-source facility that provides software tools for search based test research. The testbed tool emulates test scenarios in a controled environment using mock objects and implementing performance antipatterns.

The main contributions of this research are as follows: The presentation of a hybrid metaheuristic using Q-learning approach for use in stress tests; the development of a Testbed tool the development of a JMeter plugin for search-based tests and the automation of the stress test execution process.

8.1 Achievements

Two experiments were performed to validate the hybrid metaheuristic,two experiments were conducted to validate the testbed tool and two experiments where conducted to validate the hybridQ metaheuristic. The experiments uses genetic, algorithms, tabu search and simulated annealing.

In the first experiment, the signed-rank Wilcoxon non-parametrical procedure was used for comparing the results. The significant level adopted was 0.05. The procedure showed that there was a significant improvement in the results with the Hybrid Metaheuristic approach.

The second and third experiments ran for 17 generations. The experiments used an initial population of 4 individuals by metaheuristic. All tests in the experiment were conducted without the need of a tester,

automating the execution of stress tests with the JMeter tool. In both experiments the hybridQ metaheuristic returned individuals with higher fitness scores. However, the Hybrid metaheuristic made twice as many requests than Tabu Search to overcome it. The SA algorithm obtained the worst fitness values. The algorithm initially used a scenario with an antipattern and found neighbors that still using the antipatterns over the 17 generations of the experiment.

In the second experiment the metaheuristics converged to scenarios with an happy path, excluding the scenarios with the use of an antipatterns.The first individual has 153 users on Happy Scenario 2, 16 users on Happy Scenario 1 and a response time of 13 seconds. None of the four best individuals has one of the antipatterns used in the experiment.

In the third experiment, the metaheuristics converged to scenarios with an happy path and Tower Babel antipattern, excluding the scenarios with Unbalanced Processing antipattern. The individual with best fitness value has 121 users on Happy Scenario 2, 171 users on Happy Scenario 1 and a response time of 11 seconds. None of the four best individuals has one of the antipatterns used in the experiment.

In the fourth experiment, the whole process of stress and performance tests, which took 3 days and about 1800 executions, was carried out without the need for monitoring by a test designer. The tool automatically selected the next scenarios to be run up to the limit of six generations previously established. The small number of samples of the experiment is insufficient to give a statistical significance to the results of the Wilcoxon procedure. However, it is noted that, in four of six generations, the collaborative approach presented the best values. The experiment succeeded in finding 29 individuals whose maximum time expected by the application was obtained.

8.2 Open Issues and future works

There is a range of future improvements in the proposed approach. Also as a typical search strategy, it is difficult to ensure that the execution times generated in the experiments represents global optimum. More experimentation is also required to determine the most appropriate and robust parameters. Lastly, there is a need for an adequate termination criterion to stop the search process.

Among the future works of the research, the use of new combinatorial optimization algorithms such as very large-scale neighborhood search is one that we can highlight.

...

...

Bibliography

[1] Model-based generation of testbeds for web services. pages 266–282, 2008.

[2] Wasif Afzal, Richard Torkar, and Robert Feldt. A systematic review of search-based testing for non-functional system properties. volume 51, pages 957–976. Elsevier B.V., 2009.

[3] Jarmo T. JT Alander, Timo Mantere, and Pekka Turunen. Genetic Algorithm Based Software Testing. In *Neural Nets and Genetic Algorithms*, 1998.

[4] Stefano D I Alesio, Lionel C Briand, Shiva Nejati, and Arnaud Gotlieb. Combining Genetic Algorithms and Constraint Programming. volume 25, 2015.

[5] Aldeida Aleti, I. Moser, and Lars Grunske. Analysing the fitness landscape of search-based software testing problems. *Automated Software Engineering*, pages 1–19, 2016.

[6] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. volume 86, pages 1978–2001, 2013.

[7] Marco Anisetti, Claudio A. Ardagna, Ernesto Damiani, and Francesco Saonara. A test-based security certification scheme for web services. *ACM Transactions on the Web*, 7(2):1–41, 2013.

[8] Dejanira Araiza-Illan, Anthony G. Pipe, and Kerstin Eder. Model-based Test Generation for Robotic Software: Automata versus Belief-Desire-Intention Agents. pages 1–16, 2016.

[9] Alessandro Oliveira Arantes. Tool support for generating model-based test cases via web Valdivino Alexandre de Santiago Júnior , Nandamudi Lankalapalli Vijaykumar and Erica Ferreira de Souza. 9(1):62–96, 2014.

[10] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Improving Model-based Test Generation by Model Decomposition. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 119–130, 2015.

[11] Davide Arcelli, Vittorio Cortellessa, and Catia Trubiani. Antipattern-Based Model Refactoring for Software Performance Improvement. pages 33–42, 2012.

[12] A. Avritzer and E.J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions on Software Engineering*, 21(9):705–716, 1995.

[13] Alberto Avritzer and Brian Larson. Load Testing Software Using Deterministic State Testing. ISSTA '93, pages 82–88, New York, NY, USA, 1993. ACM.

[14] Alberto Avritzer and EJ Weyuker. Generating test suites for software load testing. pages 44–57, New York, New York, USA, 1994. ACM Press.

[15] Scott Barber. User Community Modeling Language (UCML ™) v1 . 1 for Performance Test Workloads UCML ™ Overview. pages 1–9, 1999.

[16] Marcelo De Barros and Jing Shiau. Web services wind tunnel: On performance testing large-scale stateful web services. 2007.

[17] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: overview and conceptual comparison. *ACM Computing Surveys*, 35(3):189–213, 2003.

[18] Lionel C. Briand, Yvan Labiche, and Marwa Shousha. Stress testing real-time systems with genetic algorithms. page 1021, 2005.

[19] William H Brown, Raphael C Malveau, Hays W McCormick, and Thomas J Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[20] Didier Buchs, Levi Lucio, and Ang Chen. Model checking techniques for test generation from business process models. *Reliable Software Technologies–Ada-Europe 2009*, pages 59–74, 2009.

[21] Yuhong Cai, John Grundy, and John Hosking. Synthesizing client load models for performance engineering via web crawling. *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*, page 353, 2007.

[22] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for QoS-aware service composition based on genetic algorithms. 2005.

[23] Microsoft Corporation. Performance Testing Guidance for Web Applications, November 2007.

[24] Vittorio Cortellessa and Laurento Frittella. A Framework for Automated Generation of Architectural Feedback from Software Performance Analysis. pages 171–185, 2007.

[25] MB da Silveira, EM Rodrigues, and AF Zorzo. Generation of Scripts for Performance Testing Based on UML Models. 2011.

[26] Leffingwell Dean and Widrig Don. Managing software requirements: A use case approach, 2003.

[27] S Di Alesio, S Nejati, L Briand, and A Gotlieb. Stress testing of task deadlines: A constraint programming approach. pages 158–167, 2013.

[28] Stefano Di Alesio, Shiva Nejati, Lionel Briand, and Arnaud Gotlieb. Worst-Case Scheduling of Software Tasks – A Constraint Optimization Model to Support Performance Testing. pages 813–830, 2014.

[29] Giuseppe a. Di Lucca and Anna Rita Fasolino. Testing Web-based applications: The state of the art and future trends. volume 48, pages 1172–1186, 2006.

[30] D. Draheim, J. Grundy, J. Hosking, C. Lutteroth, and G. Weber. Realistic load testing of Web applications. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, 2006.

[31] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated Software Testing: Introduction, Management, and Performance*. 1999.

[32] Eduard Paul Enoiu, Daniel Sundmark, and Paul Pettersson. Model-based test suite generation for function block diagrams using the UPPAAL model checker. *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2013*, pages 158–167, 2013.

[33] Bayo Erinle. *Performance Testing With JMeter 2.9*. 2013.

[34] Ling Fang, Takashi Kitamura, Thi Bich Ngoc Do, and Hitoshi Ohsaki. Formal model-based test for AUTOSAR multicore RTOS. *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*, pages 251–259, 2012.

[35] Dror G Feitelson. *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge University Press, 2013.

[36] Dharmalingam Ganesan, Mikael Lindvall, Stefan Hafsteinsson, Rance Cleaveland, Susanne L. Strege, and Walter Moleski. Experience Report: Model-Based Test Automation of a Concurrent Flight Software Bus. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, pages 445–454, 2016.

[37] Vahid Garousi. Traffic-aware Stress Testing of Distributed Real-Time Systems based on UML Models using Genetic Algorithms. Number August, 2006.

[38] Vahid Garousi. Empirical analysis of a genetic algorithm-based stress test technique. page 1743, 2008.

[39] Vahid Garousi. A Genetic Algorithm-Based Stress Test Requirements Generator Tool and Its Empirical Evaluation. volume 36, pages 778–797, November 2010.

[40] Gregory Gay. Challenges in Using Search-Based Test Generation to Identify Real Faults in Mockito. pages 1–6.

[41] Gregory Gay, Sanjai Rayadurgam, and Mats P.E. Heimdahl. Automated Steering of Model-Based Test Oracles to Admit Real Program Behaviors. *IEEE Transactions on Software Engineering*, PP(99), 2016.

[42] Fred Glover and Rafael Martí. Tabu Search. pages 1–16, 1986.

[43] N. Gois, P. Porfirio, A. Coelho, and T. Barbosa. Improving Stress Search Based Testing using a Hybrid Metaheuristic Approach. In *Proceedings of the 2016 Latin American Computing Conference (CLEI)*, pages 718–728, 2016.

[44] Marcelo Canário 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. 2014.

[45] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. pages 156–166. Ieee, June 2012.

[46] Amy Greenwald, Keith Hall, and R Serrano. Correlated Q-learning. Number 3, pages 84–89, 2003.

[47] Hg Gross, Bryan F Jones, and David E Eyres. Structural performance measure of evolutionary testing applied to worst-case timing of real-time systems. volume 147, pages 25–30, 2000.

[48] Emily H. Halili. *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites.* 2008.

[49] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. volume 36, pages 226–247, 2010.

[50] Anders Hessel. *Model-Based Test Case Generation for Real-Time Systems*. 2007.

[51] Robert M Hierons, Kirill Bogdanov, Jonathan P Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J H Simons, Sergiy Vilkomir, Martin R Woodward, and Hussein Zedan. Using formal specifications to support testing. volume 41, pages 1–76, 2009.

[52] Tzung-Pei Hong, Hong-Shung Wang, and Wei-Chou Chen. Simultaneously applying multiple mutation operators in genetic algorithms. volume 6, pages 439–455. Springer, 2000.

[53] B. Jones J. Wegener, K. Grimm, M. Grochtmann, H. Sthamer. Systematic testing of real-time systems. 1996.

[54] Wassim Jaziri. *Local Search Techniques: Focus on Tabu Search*. 2008.

[55] ZM Jiang. *Automated analysis of load testing results*. PhD thesis, 2010.

[56] Gyu Baek Kim. A method of generating massive virtual clients and model-based performance test. *Proceedings - International Conference on Quality Software*, 2005:250–254, 2005.

[57] Chris Lenz, J Chimiak-Opoka, and Ruth Breu. Model-Driven Testing of SOA-based Software. ... *of the SEMSOA Workshop on Software* ..., 2007.

[58] William E. Lewis, David Dobbs, and Gunasekaran Veerapillai. *Software testing and continuous quality improvement*. 2005.

[59] Qi Luo, Aswathy Nair, Mark Grechanik, and Denys Poshyvanyk. FOREPOST: finding performance problems automatically with feedback-directed learning software testing. pages 1–51, 2015.

[60] Michel Mamrot, Stefan Marchlewitz, Jan Peter Nicklas, Petra Winzer, Thomas Tetzlaff, Philipp Kemper, and Ulf Witkowski. Model-Based Test and Validation Support for Autonomous Mechatronic Systems. *Proceedings - 2015 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2015*, pages 701–706, 2016.

[61] Thomas Stützle Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[62] Alexander Pretschner Mark Utting and Bruno Legeard. A taxonomy of model-based testing approaches. volume 24, pages 297–312, 2012.

[63] Daniel A Menascé and George Mason. TPC-W : A Benchmark for E-commerce. Number June, pages 1–6, 2002.

[64] Petros Nicopolitidis Mohammad S. Obaidat and Faouzi Zarai. *Modeling and Simulation of Computer Networks and Systems Methodologies and Applications*.

[65] Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. "O'Reilly Media, Inc.", 1st edition, January 2009.

[66] F. Mueller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. 1998.

[67] Conference Paper. Performance Issues in Statistical Testing. (April 2006), 2014.

[68] Massimiliano Di Penta, Gerardo Canfora, and Gianpiero Esposito. Search-based testing of service level agreements. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1090–1097, 2007.

[69] William E. Perry. *Effective methods for software testing*. 2004.

[70] Hartmut Pohlheim, Mirko Conrad, and Arne Griep. Evolutionary Safety Testing of Embedded Control Software by Automatically Generating Compact Test Data Sequences. Number 724, pages 804—-814, 2005.

[71] Jakob Puchinger and R Raidl. Combining Metaheuristics and Exact Algorithms in Combinatorial Optimization : A Survey and Classification. volume 3562, pages 41–53, 2005.

[72] P. Puschner and R. Nossal. Testing the results of static worst-case execution-time analysis. 1998.

[73] Günther R Raidl, Jakob Puchinger, and Christian Blum. Metaheuristic hybrids. In *Handbook of metaheuristics*, pages 469–496. Springer, 2010.

[74] R Raidl. A Unified View on Hybrid Metaheuristics. pages 1–12, 2006.

[75] Irum Rauf, Muhammad Zohaib Z Iqbal, and Zafar I Malik. Model Based Testing of Web Service Composition Using UML Profile. *2nd Workshop on Model-based Testing in Practice, MOTIP 2009*, 2009.

[76] Elder M Rodrigues, Rodrigo S Saad, Flavio M Oliveira, Leandro T Costa, Maicon Bernardino, and Avelino F Zorzo. Evaluating Capture and Replay and Model-based Performance Testing Tools: An Empirical Comparison. *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 9:1—-9:8, 2014.

[77] Jose Lorenzo San Miguel and Shingo Takada. GUI and Usage Model-based Test Case Generation for Android Applications with Change Analysis. *Proceedings of the 1st International Workshop on Mobile Development*, pages 43–44, 2016.

[78] Corey Sandler, Tom Badgett, and TM Thomas. The Art of Software Testing. page 200. John Wiley & Sons, September 2004.

[79] Christopher Schaefer, Hyunsook Do, and Brian M. Slator. Crushinator: A framework towards game-independent testing. *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, pages 726–729, 2013.

[80] Mahesh Shirole and Rajeev Kumar. UML behavioral model based test case generation. *ACM SIGSOFT Software Engineering Notes*, 38(4):1, 2013.

[81] Marwa Shousha. *Performance Stress Testing of Real-Time Systems Using Genetic Algorithms*. PhD thesis, Carleton University Ottawa, 2003.

[82] Connie U. Smith and Lloyd G. Williams. Software performance antipatterns. pages 127–136, 2000.

[83] Connie U Smith and Lloyd G Williams. More New Software Performance AntiPatterns: EvenMore Ways to Shoot Yourself in the Foot. pages 717–725, 2003.

[84] C.U. Smith and L.G. Williams. Software Performance AntiPatterns; Common Performance Problems and their Solutions. volume 2, pages 797–806, 2002.

[85] Adepu Sridhar, D. Srinivasulu, and Durga Prasad Mohapatra. Model-based test-case generation for Simulink/Stateflow using dependency graph approach. *Proceedings of the 2013 3rd IEEE International Advance Computing Conference, IACC 2013*, pages 1414–1419, 2013.

[86] Michael O Sullivan, Siegfried Vössner, Joachim Wegener, and Daimler-benz Ag. Testing Temporal Correctness of Real-Time Systems — A New Approach Using Genetic Algorithms and Cluster Analysis —. pages 1–20, 1998.

[87] Richard S. Sutton and Andrew G. Barto. Reinforcement learning. volume 3, page 322, 2012.

[88] El-Ghazali Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009.

[89] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*, volume 53. 2013.

[90] N J Tracey, J a Clark, and K C Mander. Automated Programme Flaw Finding using Simulated Annealing. 1998.

[91] Nigel James Tracey. *A search-based automated test-data generation framework for safety-critical software*. PhD thesis, Citeseer, 2000.

[92] G Trent and M Sake. WebSTONE: The first generation in {HTTP} server benchmarking. 1995.

[93] Via Vetoio. PhD Thesis in Computer Science Automated generation of architectural feedback from software performance analysis results Catia Trubiani. 2011.

[94] Christian Vogele, André van Hoorn, Eike Schulz, Wilhelm Hasselbring, and Helmut Krcmar. WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction???a model-driven approach for session-based application systems. Number October, pages 1–35. Springer Berlin Heidelberg, 2016.

[95] Xingen Wang, Bo Zhou, and Wei Li. Model-based load testing of web applications. volume 36, pages 74–86, 2013.

[96] J Wegener and M Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. volume 15, pages 275–298, 1998.

[97] Joachim Wegener, Harmen Sthamer, Bryan F Jones, and David E Eyres. Testing real-time systems using genetic algorithms. volume 6, pages 127–135, 1997.

[98] Harmen Wegener, Joachim and Pitschinetz, Roman and Sthamer. Automated Testing of Real-Time Tasks. 2000.

[99] Alexander Wert, Jens Happe, and Lucia Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. Number May, pages 552–561, 2013.

[100] Alexander Wert, Marius Oehler, Christoph Heger, and Roozbeh Farahbod. Automatic detection of performance anti-patterns in inter-component communications. pages 3–12, 2014.

[101] S Wieczorek, A Stefanescu, and A Roth. Model-Driven Service Integration Testing - A Case Study. *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pages 292–297, 2010.

[102] Hongwei Zeng Xinying Cai. Model-based Test Generation for Software Product Line. 2007.