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

by

Francisco Nauber Bernardo Gois

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Science in Computer Science and Engineering

at the

Universidade de Fortaleza

February 2017

. . . . . .
Dsc. Pedro Porfirio Muniz de Farias
Advisor - Universidade de Fortaleza

. . . . . .
Dsc. Andre Coelho
Co-Advisor - Universidade de Fortaleza

. . . . . .
Dsc. Pedro
Co-External Examiner - Universidade Federal do Piaui

Accepted by . . . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

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

by

Francisco Nauber Bernardo Gois

Submitted to the Department of Electrical Engineering and Computer Science
on February 16, 2017, in partial fulfillment of the
requirements for the degree of
Doctor of Science in Computer Science and Engineering

Abstract

Some software systems must respond to thousands or millions of concurrent requests. These systems must be properly tested to ensure that they can function correctly under the expected load. A common use of stress testing is to find test scenarios that produce execution times that violate the timing constraints specified. In this context, search-based testing is seen as a promising approach for verifying timing constraints. In this thesis, We proposed hybrid metaheuristic approach that uses genetic algorithms, simulated annealing, and tabu search algorithms in a collaborative model using Q-Learning to improve stress search-based testing and automation. A tool named IAdapter, a JMeter plugin used for performing search-based stress tests, was developed. Four experiments were conducted to validate the proposed approach.

Thesis Supervisor: Pedro Porfirio Muniz de Farias
Title: Associate Professor

Acknowledgments

This is the acknowledgements section. You should replace this with your own acknowledgements.

Related Publications

The following publications are related to this thesis:

- **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 [31].

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many systems must support concurrent access by hundreds or thousands of users. Failure to providing scalable access to users may results in catastrophic failures and unfavorable media coverage [40].

The explosive growth of the Internet has contributed to the increased need for 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 to fix them [45].

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 [22] [40].

Stress testing determines the responsiveness, throughput, reliability, or scalability of a system under a given workload. The quality of the results of applying a given load testing to a system is closely linked to the implementation of the workload strategy. The performance of many applications depends on the load applied under different conditions. In some cases, performance degradation and failures arise only in stress conditions [27] [40].

A stress test uses a set of workloads that consist of many types of usage scenarios and a combination of different numbers of users. A load is typically based on an operational profile. Different parts of an application should be tested under various parameters and stress conditions [9]. The correct application of a stress test should cover most parts of an

application above the expected load conditions[22].

Fig. 1-1 shows an example of a system under assessment with three pages (the main page, profile page, and search page) and six possible users. From the combinations of users and application pages, various scenarios can be created, such as scenarios 1 and 2 shown in the figure. The first scenario presents a test that has passed, and the second scenario presents a test that has an HTTP 404 error.

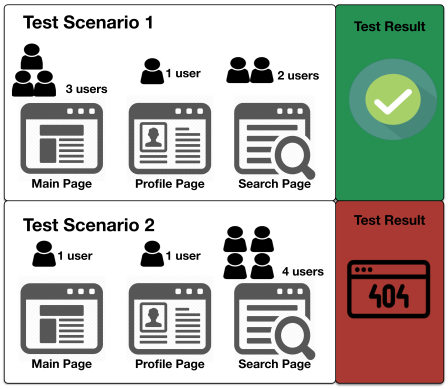


Figure 1-1: Possible test scenarios for a hypothetical application

A stress test usually lasts for several hours or even a few days and only tests a limited number of workloads. The major challenge is to find the workloads that expose a major number of errors and to discover the maximum number of users supported by an application under testing [10].

Search-based testing is seen as a promising approach to verifying 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 [57].

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 [37] [18].

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 [62].

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] [28].

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 [33].

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. Most research studies are limited to making prototype. The presented research work is distinguished from others by having extend a industrial tool using a hybrid approach and q-learning [2].

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 [41].

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 [33].

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 Overview

In this section, we present an overview of the works presented in this thesis. This thesis has five main chapters

1.4 Thesis Contributions

The major contributions of this thesis are:

- Hybrid Metaheuristic Approach [31]: We present a hybrid metaheuristic approach that uses Genetic Algorithms, Simulated Annealing and Tabu Search in a collaborative mode.
- Q-Learning Metaheuristic Approach:
- IAdapter JMeter Plugin:
- Testbed Tool:

1.5 Thesis Organization

Chapter 2

A Survey on Stress Testing Software Systems

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 [53] [16].

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 [21]. The purpose of this assessment is to demonstrate that the system reaches its performance objectives [53]. Term often used interchangeably with "stress" and "load" testing. Ideally "performance" testing is defined in requirements documentation or QA or Test Plans [41].

In a load testing, the system is evaluated at predefined load levels [21]. 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 [45]. 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 [53] [41], 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 [21] [45].

This chapter surveys the state of the art literature in stress testing research. This survey will be useful for stress testing practitioners and software engineering researchers with interests in testing and analyzing software systems. We proposed the following four research questions:

- How is a proper stress designed?
- How is a stress test executed and automated?
- What are the main problems found by stress tests?
- How are the stress tests results analysed?

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.

2.1 Background

Stress testing is boundary testing. Some of the resources that stress testing subjects to heavy loads include [41]:

- Memory
- Networks
- Transaction queues
- Transaction schedulers
- User of the system

The following are the suggested steps for stress testing [41]:

- Perform simple multitask tests.

- After the simple stress defects are corrected, stress the system to the breaking point.

- Perform the stress tests repeatedly for every development spiral.

While load testing is the process of assessing non-functional quality related problems under load. Performance testing is used to measure and/or evaluate performance related aspects (e.g., response time, throughput and resource utilizations) of algorithms, designs/architectures, modules, configurations, or the overall systems. Stress tests puts a system under extreme conditions to verify the robustness of the system and/or detect various functional bugs (e.g., memory leaks and deadlocks) [2]. The next subsections present details about the stress test process, automated stress test tools and the stress test results.

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. 2-1 shows a common Load, Performance and Stress test process [40].

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 [40].

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[40].

The core activities in conducting an usual Load, Performance and Stress tests are [23]:

- 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.

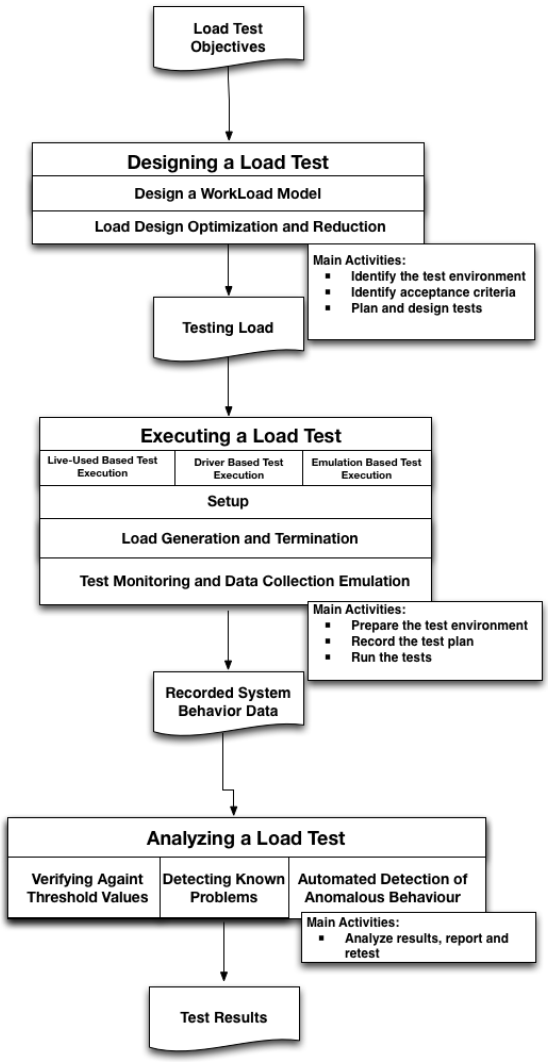


Figure 2-1: Load, Performance and Stress Test Process [40][23]

- 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 [16]. 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 [41].

- Prepare the test environment: configure the test environment, tools, and resources necessary to conduct the planned test scenarios.
- Record the test plan: record the planned test scenarios using a testing tool.
- Run the tests: Once recorded, execute the test plans under light load and verify the correctness of the test scripts and output results.
- Analyze results, report, and retest: examine the results of each successive run and identify areas of bottleneck that need addressing.

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

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 [45].

The term workload represents the size of the demand that will be imposed on the application under test in an execution. The metric used for measure a workload is dependent on the application domain, such as the length of the video in a transcoding application for multimedia files or the size of the input files in a file compression application [24] [45] [32].

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 [24] [45].

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 [24] [45].

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 [21].

In descriptive models, one finds different levels of abstraction on the one hand and different levels of fidelity to the original data on the other hand. The most strictly faithful models try to mimic the data directly using the statistical distribution of the data. The most common strategy used in descriptive modeling is to create a statistical model of an observed workload (Fig. 2-2). This model is applied to all the workload attributes, e.g., computation, memory usage, I/O behavior, communication, etc. [21]. Fig. 2-2 shows a simplified workflow of a descriptive model. The workflow has six phases. In the first phase, the user uses the system in the production environment. In the second phase, the tester collects the user's data, such as logs, clicks, and preferences, from the system. The third phase consists in developing a model designed to emulate the user's behavior. The fourth phase is made up of the execution of the test, emulation of the user's behavior, and log gathering.

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

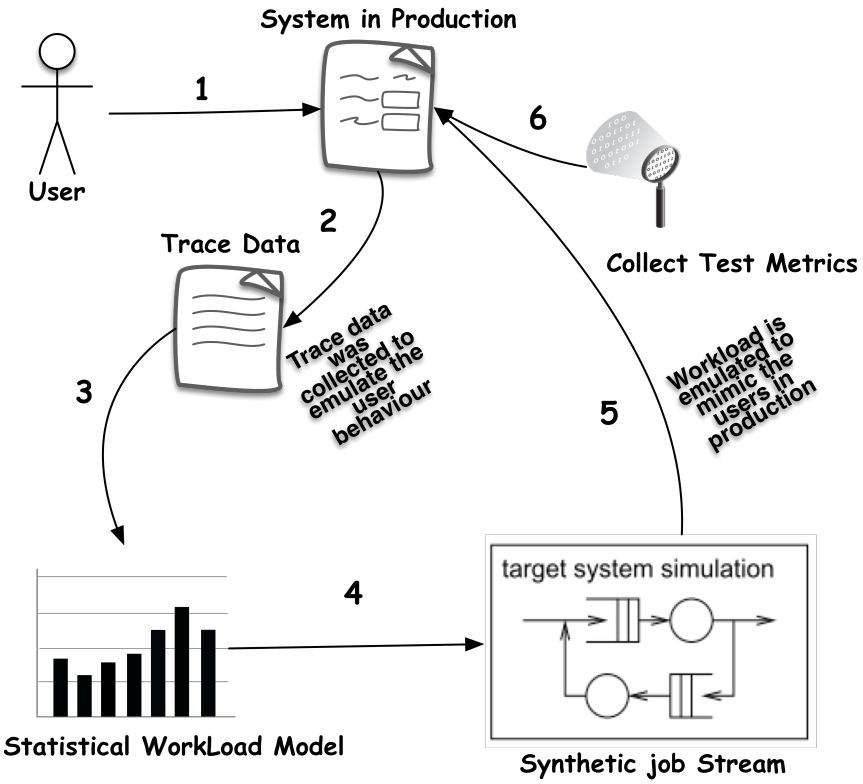


Figure 2-2: Workload modeling based on statistical data [21]

desired conditions [21]. 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. 2-3).

Both load model have their advantages and disadvantages. In general, loads resulting from realistic-load based design techniques (Descriptive models) can be used to detect both functional and non-functional problems. However, the test durations are usually longer and the test analysis is more difficult. Loads resulting from fault-inducing load design techniques (Generative models) take less time to uncover potential functional and non-functional problems, the resulting loads usually only cover a small portion of the testing objectives [40]. 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

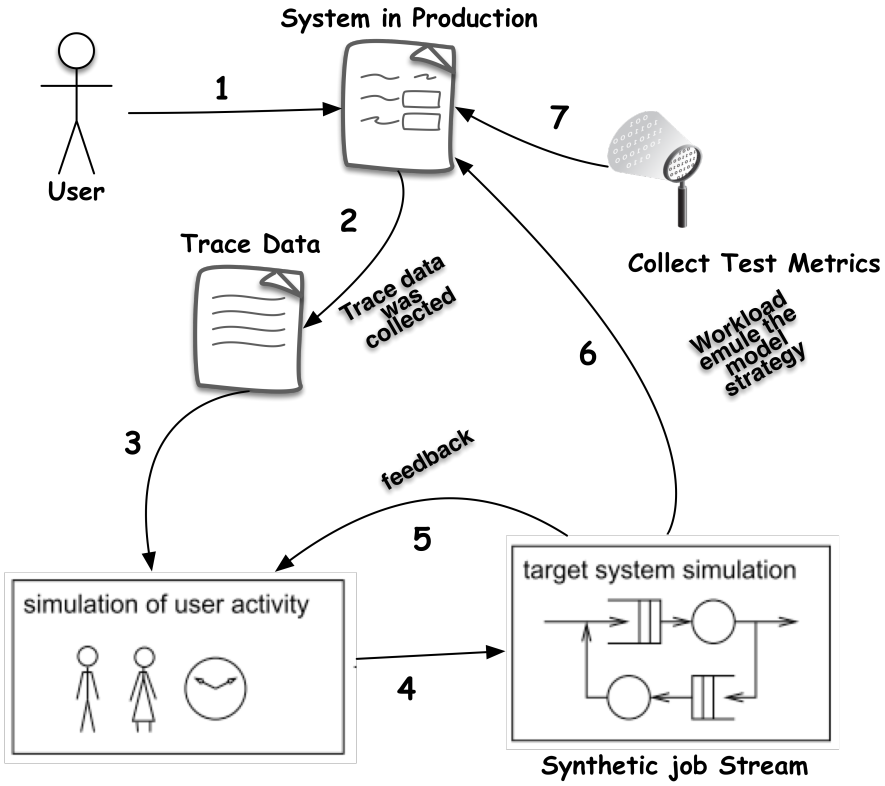


Figure 2-3: Workload modeling based on the generative model [21]

learning testing system that learns rules from system execution [**?**] [68].

- Search-based Stress testing:

2.2.1 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 [42] [1].

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 [63]. The Fig. 2-4 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.

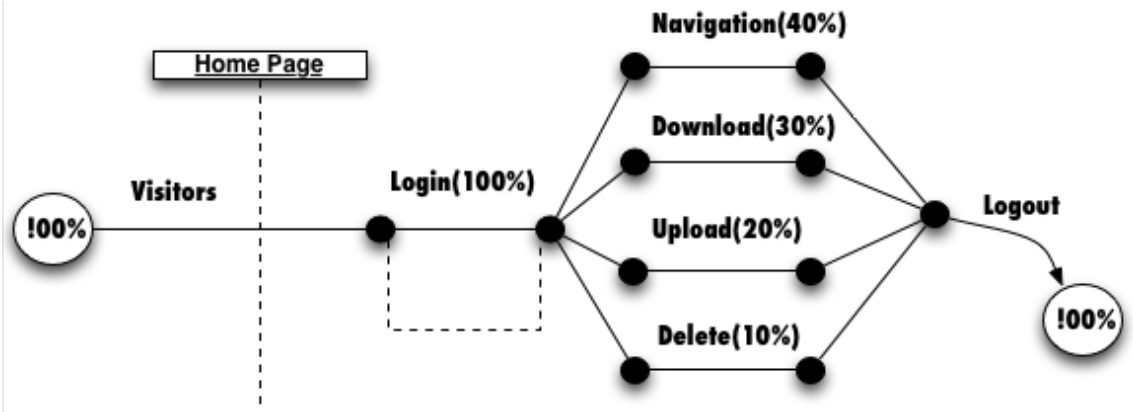


Figure 2-4: User community modeling language [63]

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. Form-oriented 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. 2-5 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 [22].

One way to capture the navigational pattern within a session is through the Customer Behavior Model Graph (CBMG). Figure 2-6 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 [43] [40] [44].

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. 2-7).

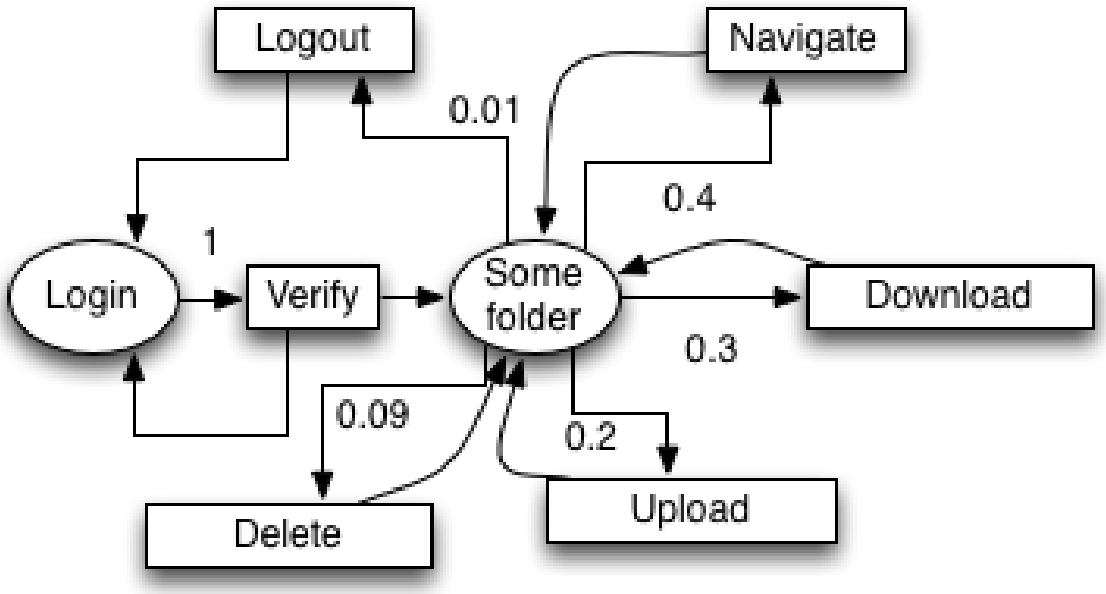


Figure 2-5: Stochastic Formcharts Example [22] [63]

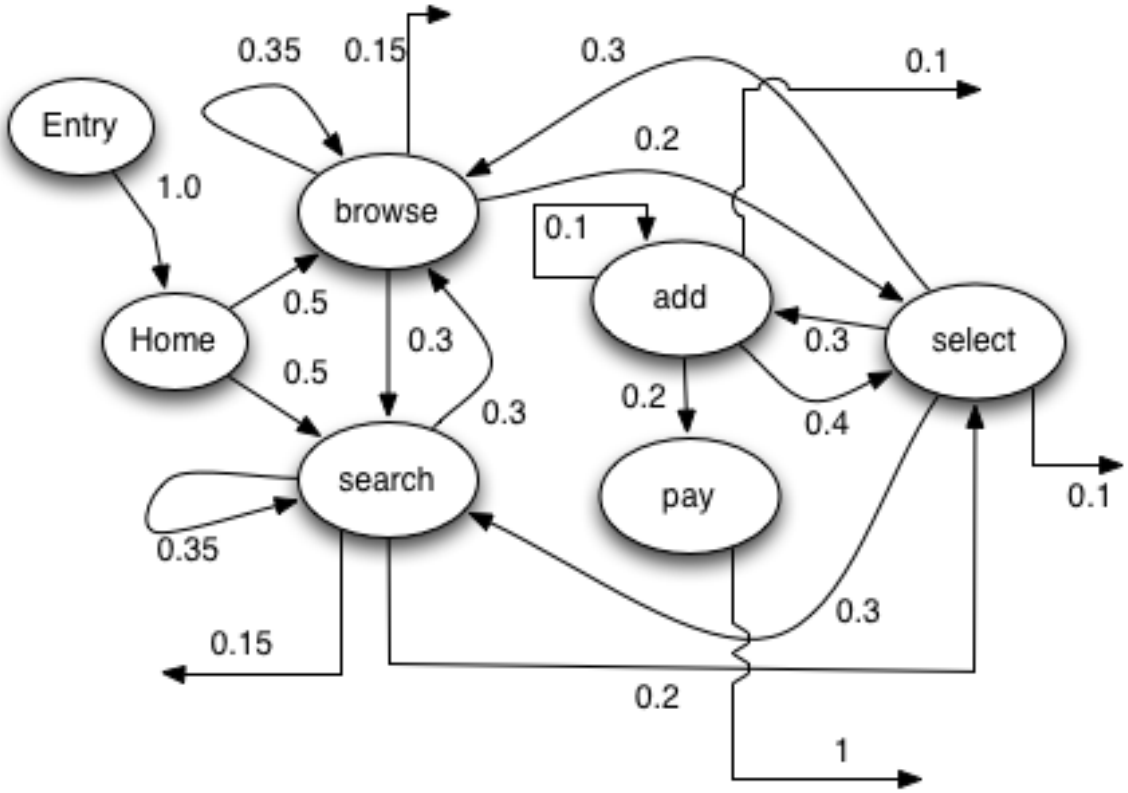


Figure 2-6: Example of a Customer Behavior Model Graph (CBMG)

2.2.2 Feedback-Directed Learning 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

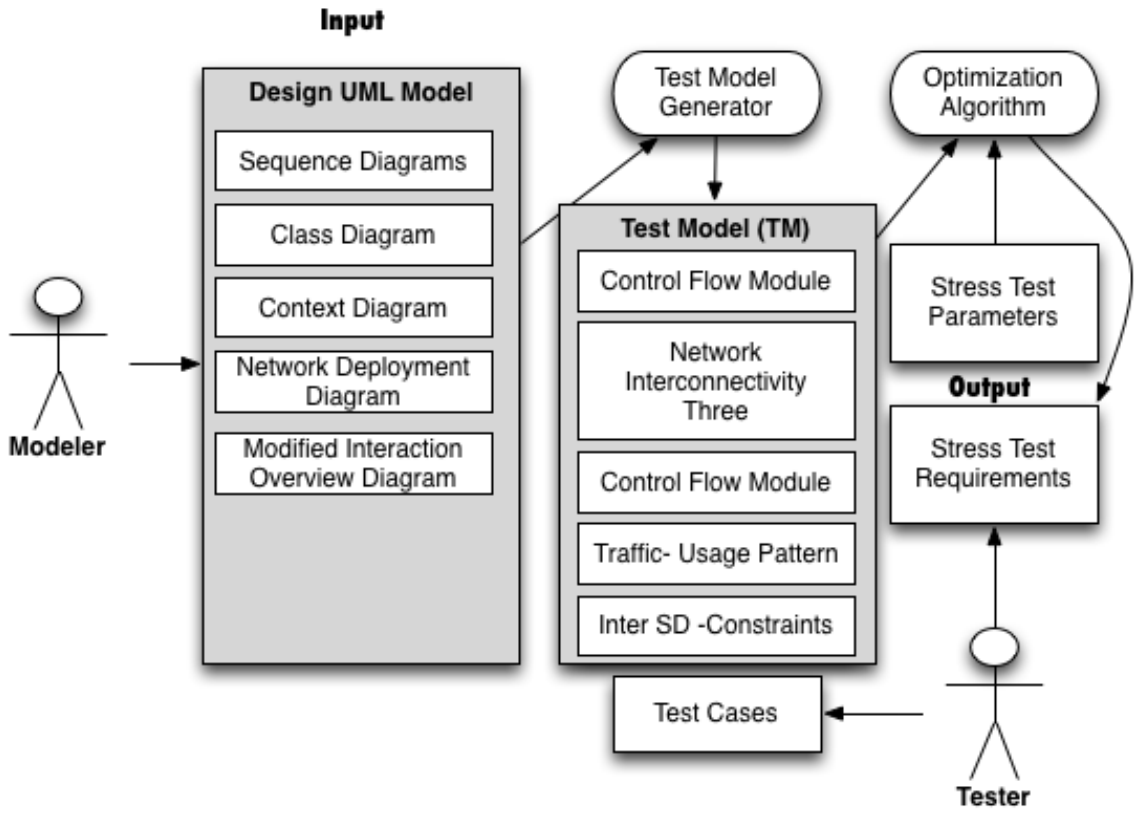


Figure 2-7: Model-based stress test methodology

these learned rules to select test input data automatically to find more performance problems in applications when compared to exploratory random performance testing [33].

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. 2-8 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.

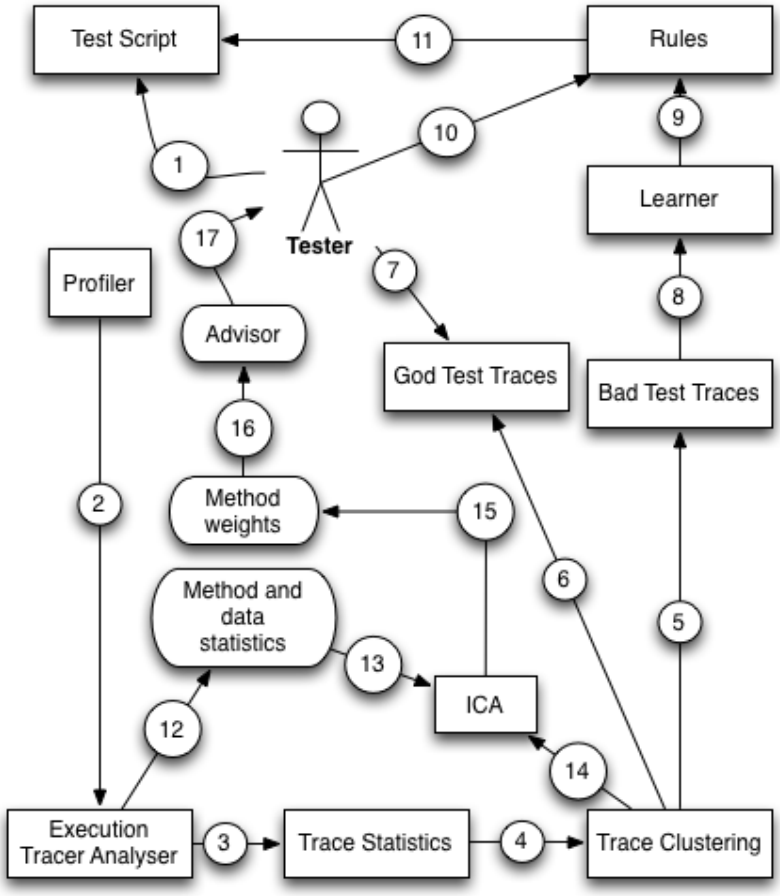


Figure 2-8: The architecture and workflow of FOREPOST

2.3 Research Question 2: How is a stress test executed and automated?

The stress test execution consists of deploy the system and steup 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 [45][40]:

- 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 [23] [44] [63]. There are three categories of load drivers:

2.3.1 Load Test Tools

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. Automated Test Tool typically have the following components [45]:

- 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.

There are several tool to execution of Stress testing. In this tools, the procedure 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 [33].

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 behavior partially [44] [43].

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 off-line, 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. 2-9). 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 [44].

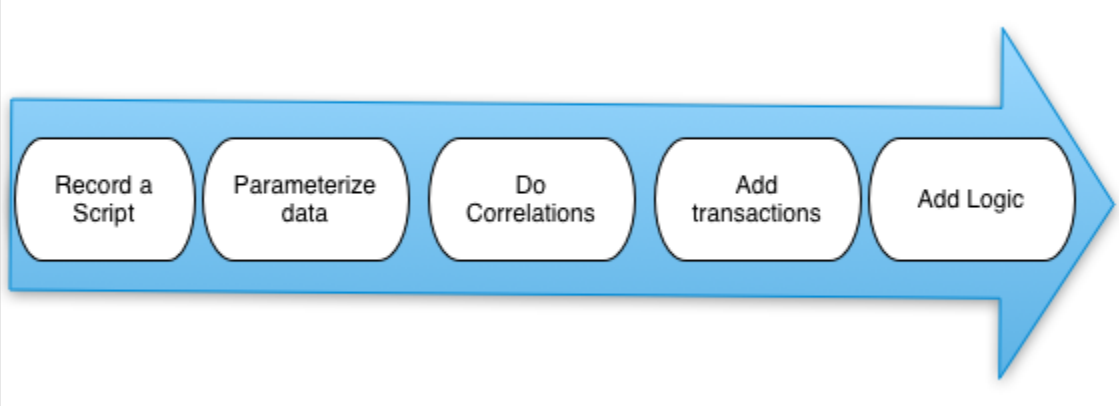


Figure 2-9: Load Runner Scripting

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 [44].

Apache JMeter

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 [35] [23].

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.

2.4 Research Question 3: 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. 2-10 shows the symptoms of known performance problems [67].

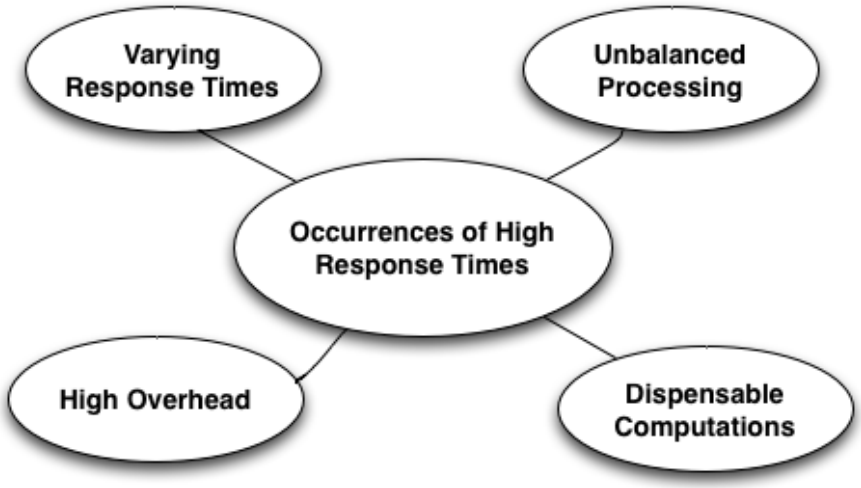


Figure 2-10: Symptoms of know performance problems [67].

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 [14]. The table 2.1 present some of the most common performance antipatterns.

Table 2.1: 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	

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 [17] [54].

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 [54]. The Figures 2-11 and 2-12 describes an hypotetical system with a BLOB problem: The Fig. 2-11 presents a sample where the Blob class uses the features A,B,C,D,E,F and G of the hypotetical system; The Fig. 2-12 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 [62][67].

Unbalanced Processing it's characterises for one scenario where a specific class of re-

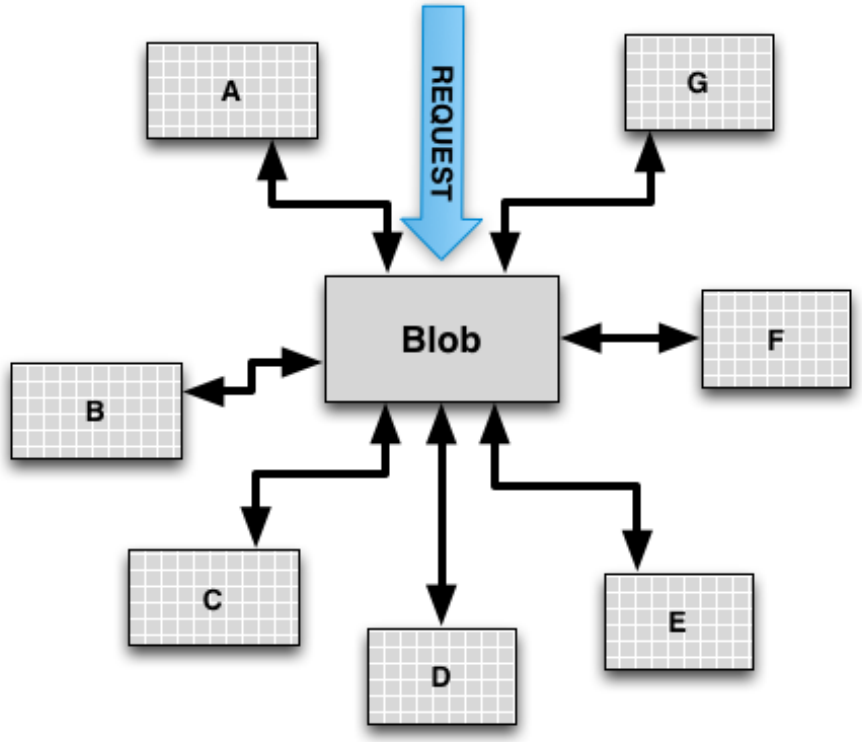


Figure 2-11: The God class[67].

quests 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. 2-13 shows a sample of the Unbalanced Processing. In The Fig. 2-13, 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. 2-14 describes a software S with a Pipe and Filter Architectures problem: (a) Static View, there

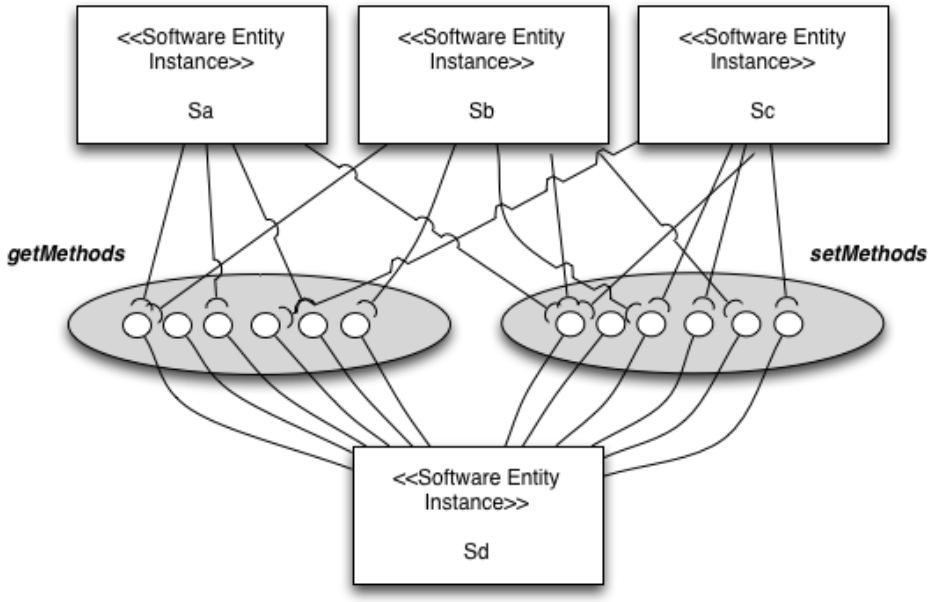


Figure 2-12: The God class[62].

is a software entity instance, e.g. Sa, offering an operation (operation x) whose resource demand (computation = $compOpx, storage = $storOpx, bandwidth = $bandOpx) is quite high; (b) Dynamic View, the operation opx is invoked in a service and the throughput of the service ($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. 2-15 describes a software S with a Extensive Processing problem: (a) Static View, there is a software entity instance, e.g. Sa, offering two operations (operation x, operation y) whose resource demand is quite unbalanced, since opx has a high demand (computation = $compOpx, storage = $storOpx, bandwidth = $bandOpx), whereas opy has a low demand (computation = $compOpy, storage = $storOpy, bandwidth = $bandOpy); (b) Dynamic View, the operations opx and opy are alternatively invoked in a service and the response time of the service ($RT(S)) is larger than the required one [62].

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 [56] [55]. 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

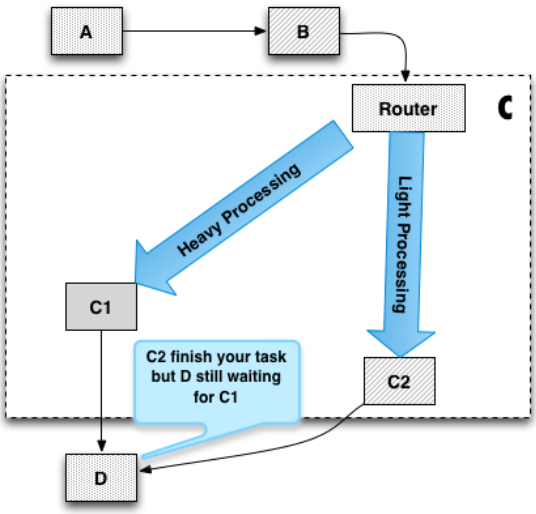


Figure 2-13: Unbalanced Processing sample [67].

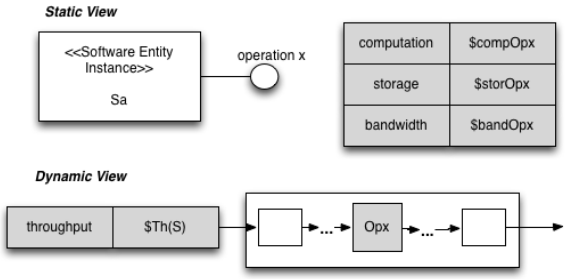


Figure 2-14: Pipe and Filter sample [62]

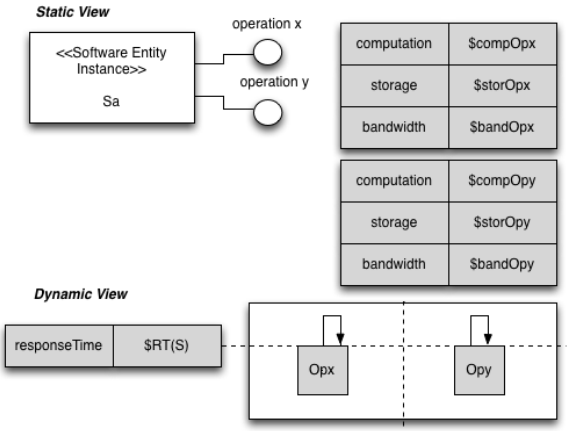


Figure 2-15: Extensive Processing sample [62].

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 [68]. The Fig. 2-16 shows a software S with a Circuitous Treasure Hunt problem: (a) Static View, there is a software entity instance e.g. Sa, retrieving information from the database; (b) Dynamic View, the software S generates a large number of database calls by performing several queries up to the final operation [62].

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

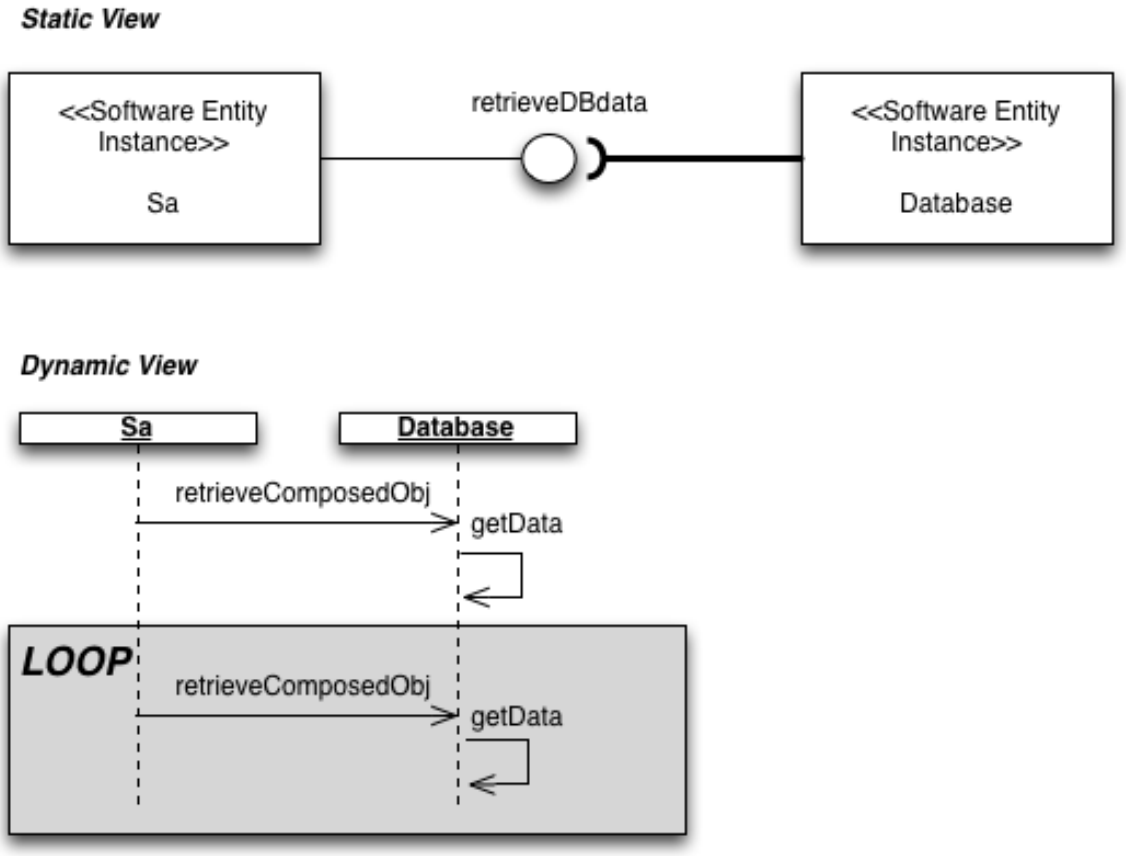


Figure 2-16: Circuitous Treasure Hunt sample [62]

both [7]. 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: (a) Static View, there is a software entity instance, e.g. Sa, retrieving some information from several instances (Remote Software 1, . . . , Remote Software n); (b) Dynamic View, 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 [62].

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 [55]. Fig. shows a system with a Tower of Babel problem: (a) Static View, there are some software entity instances, e.g. Sa, Sb, . . . , Sn; (b) Dynamic View, the software instances

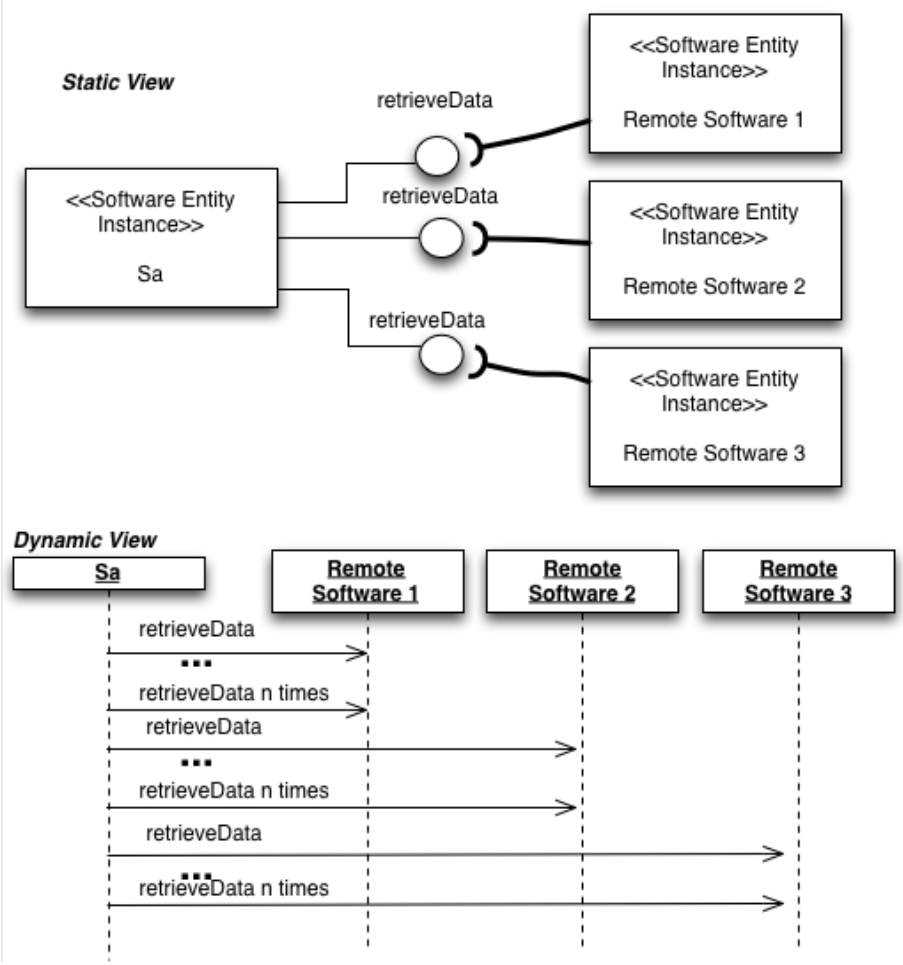


Figure 2-17: Empty Semi Trucks sample [62].

Sd performs many times the translation of format for communicating with other instances [62].

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: (a) Static View, there is a software entity instance with a capac-

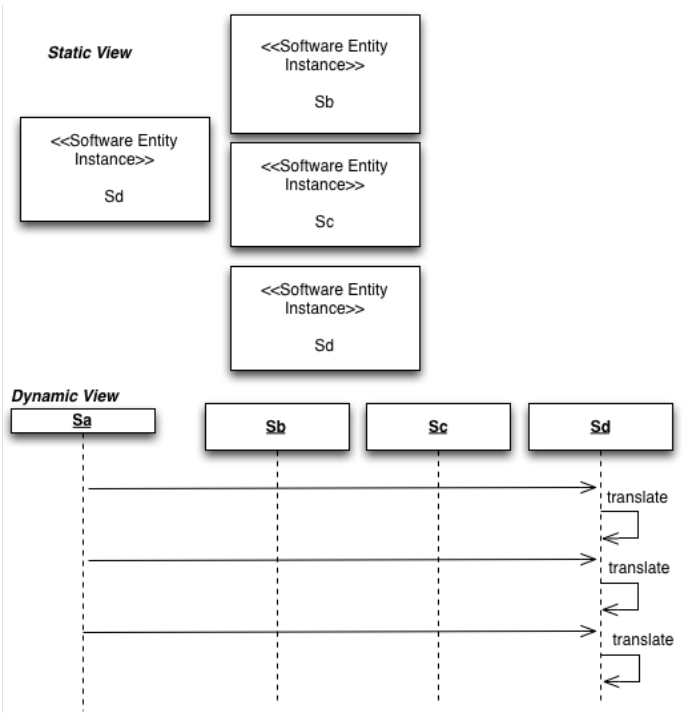


Figure 2-18: Tower of Babel sample [62]

ity of managing $poolSize threads; (b) Dynamic View, 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 [62].

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 [56] [55].

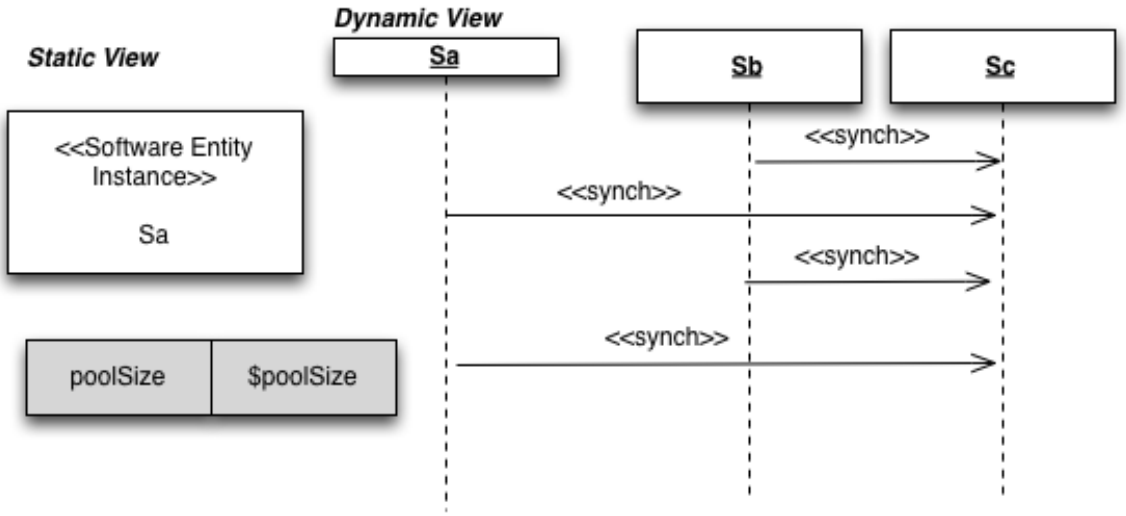


Figure 2-19: One-Lane Bridge sample [62].

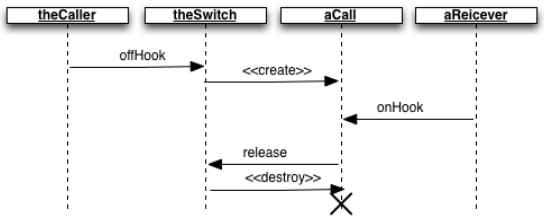


Figure 2-20: Excessive Dynamic Allocation.

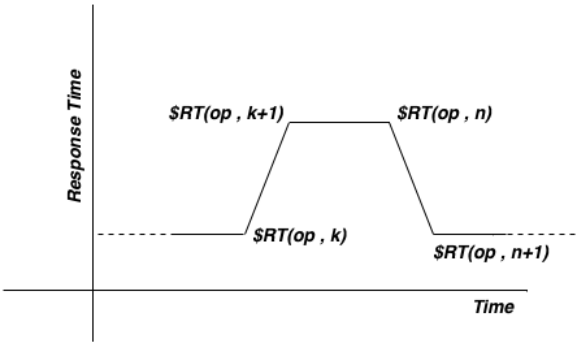


Figure 2-21: Traffic Jam Response Time [62].

The Fig. 2-20 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 2-21 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 [62].

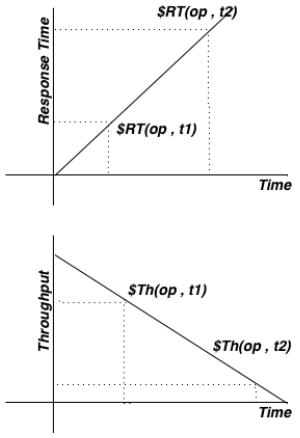


Figure 2-22: The Ramp sample [62].

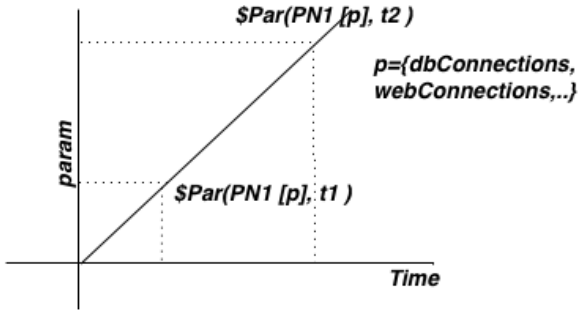


Figure 2-23: More is Less sample [62].

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 [68] [55]. Fig. 2-22 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. 2-23

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.

2.5 Research Question 4: How are the stress tests results analysed?

The system behavior recorded during the stress test execution phase needs to be analyzed to determine if there are any load-related functional or non-functional problems [40].

There can be many formats of system behavior like resource usage data or end-to-end response time, which is recorded as response time for each individual request. These types of data need to be processed before comparing against threshold values. A proper data summarization technique is needed to describe these many data instances into one number. Some researchers advocate that the 90-percentile response time is a better measurement than the average/medium response time, as the former accounts for most of the peaks, while eliminating the outliers [40].

Chapter 3

Search-Based Stress Testing

The goal of this Chapter is to describe Search-Based Testing, define recurring solutions in search-based testing

3.1 Introduction

Search-based software engineering (SBSE) is the application of optimization techniques in solving software engineering problems [1,2]. The applicability of optimization techniques in solving software engineering problems is suitable as these problems frequently encounter competing constraints and require near optimal solutions [2] [36].

Search Based Software Testing (SBST) is the sub-area of Search Based Software Engineering concerned with software testing. Search-based software testing is the application of metaheuristic search techniques to generate software tests. SBSE uses computational search techniques to tackle software engineering problems, typified by large complex search spaces. SBSE derives test inputs for a software system with the goal of improving various criteria. The test adequacy criterion is transformed into a fitness function and a set of solutions in the search space are evaluated with respect to the fitness function using a metaheuristic search technique [2] [6] [36].

Stress search-based testing is a application of SBST where the main goal it is to find test scenarios that produce execution times that exceed the timing constraints specified. If a temporal error is found, the test was successful [57].

3.2 Search-Based Testing

Search-Based Testing is the process of automatically generating test according to a test adequacy criterion,encoded as a fitness function, using search-based optimization algorithms, which are guided by a 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 [37].

Search–Based Testing uses metaheuristic algorithms to automate the generation of test inputs that meet a test adequacy criterion. Many algorithms have been considered in the past, including Simulated Annealing, Parallel Evolutionary Algorithms [4],Evolution Strategies [1], Estimation of Distribution Algorithms [48], Scatter Search [11], Particle Swarm Optimization [57], Tabu Search [13] and the Alternating Variable Method [29]. An advantage of meta-heuristic algorithms is that they are widely applicable to problems that are infeasible for analytic approaches. All one has to do is come up with a representation for candidate solutions and an objective function to evaluate those solution [8].

One of the most popular search techniques used in SBST belong to the family of Evolutionary Algorithms in what is known as Evolutionary Testing. Evolutionary Algorithms represent a class of adaptive search techniques based on natural genetics and Darwin's theory of evolution. They are characterized by an iterative procedure that works in parallel on a number of potential solutions to a problem. Figure 3-1 shows the cycle of an Evolutionary Algorithm when used in the context of Evolutionary Testing [8].

First, a population of possible solutions to a problem is created, usually at random. Starting with randomly generated individuals results in a spread of solutions ranging in fitness because they are scattered around the search–space. Next, each individual in the population is evaluated by calculating its fitness via a fitness function. The principle idea of an Evolutionary Algorithm is that fit individuals survive over time and form even fitter individuals in future generations. Selected individuals are then recombined via a crossover operator. After crossover, the resulting offspring individuals may be subjected to a mutation operator. The algorithm iterates until a global optimum is reached or another stopping condition is fulfilled [8].

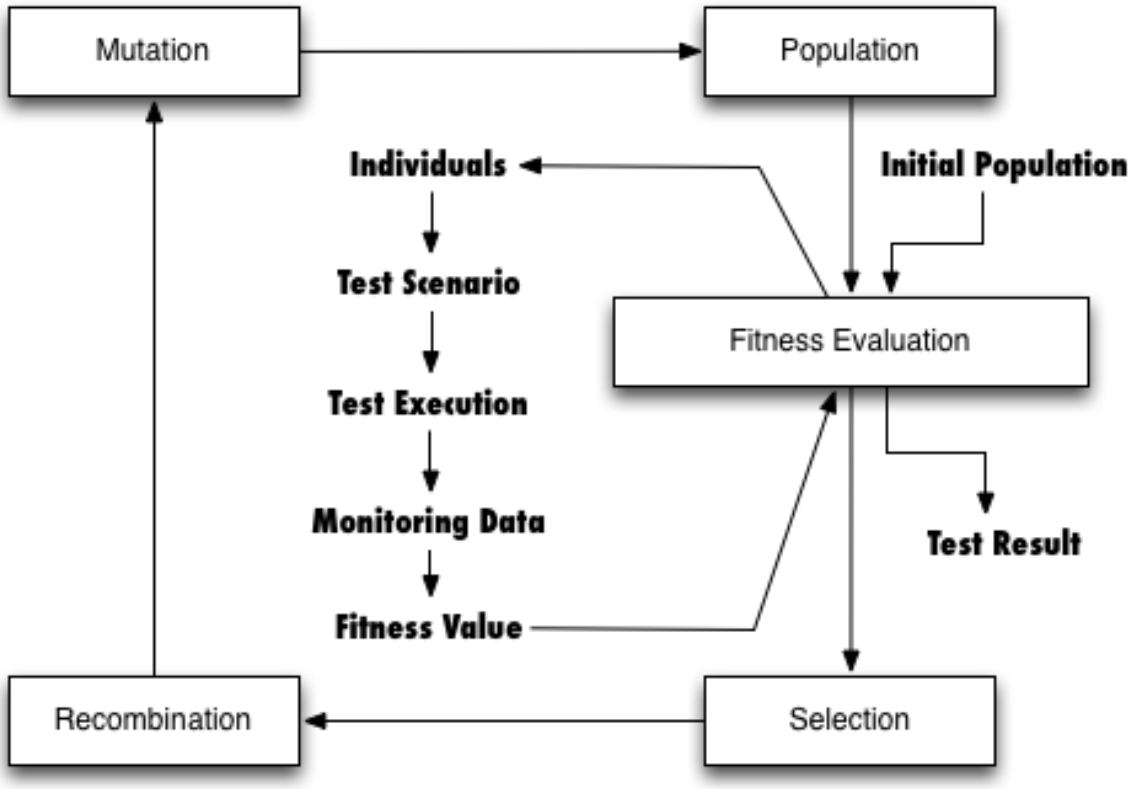


Figure 3-1: Evolutionary Algorithm Search Based Test Cycle[8].

3.3 Non-functional Search-Based Testing

SBST has made many achievements, and demonstrated its wide applicability and increasing uptake. Nevertheless, there are pressing open problems and challenges that need more attention like to extend SBST to test non-functional properties, a topic that remains relatively under-explored, compared to structural testing. There are many kinds of non-functional search based tests [2]:

- Execution time: The application of evolutionary algorithms to find the best and worst case execution times (BCET, WCET).
- Quality of service: uses metaheuristic search techniques to search violations of service level agreements (SLAs).
- Security: apply a variety of metaheuristic search techniques to detect security vulnerabilities like detecting buffer overflows.

- Usability: concerned with construction of covering array which is a combinatorial object.

- Safety: Safety testing is an important component of the testing strategy of safety critical systems where the systems are required to meet safety constraints.

A variety of metaheuristic search techniques are found to be applicable for non-functional testing including simulated annealing, tabu search, genetic algorithms, ant colony methods, grammatical evolution, genetic programming and swarm intelligence methods. The Fig. 3-2 shows a comparison between the range of metaheuristcs and the type of non-functional search based test.The Data comes from Afzal et al. [2]. Afzal's work adds to some of the latest research in this area ([25] [27] [19] [20] [5] [31]).

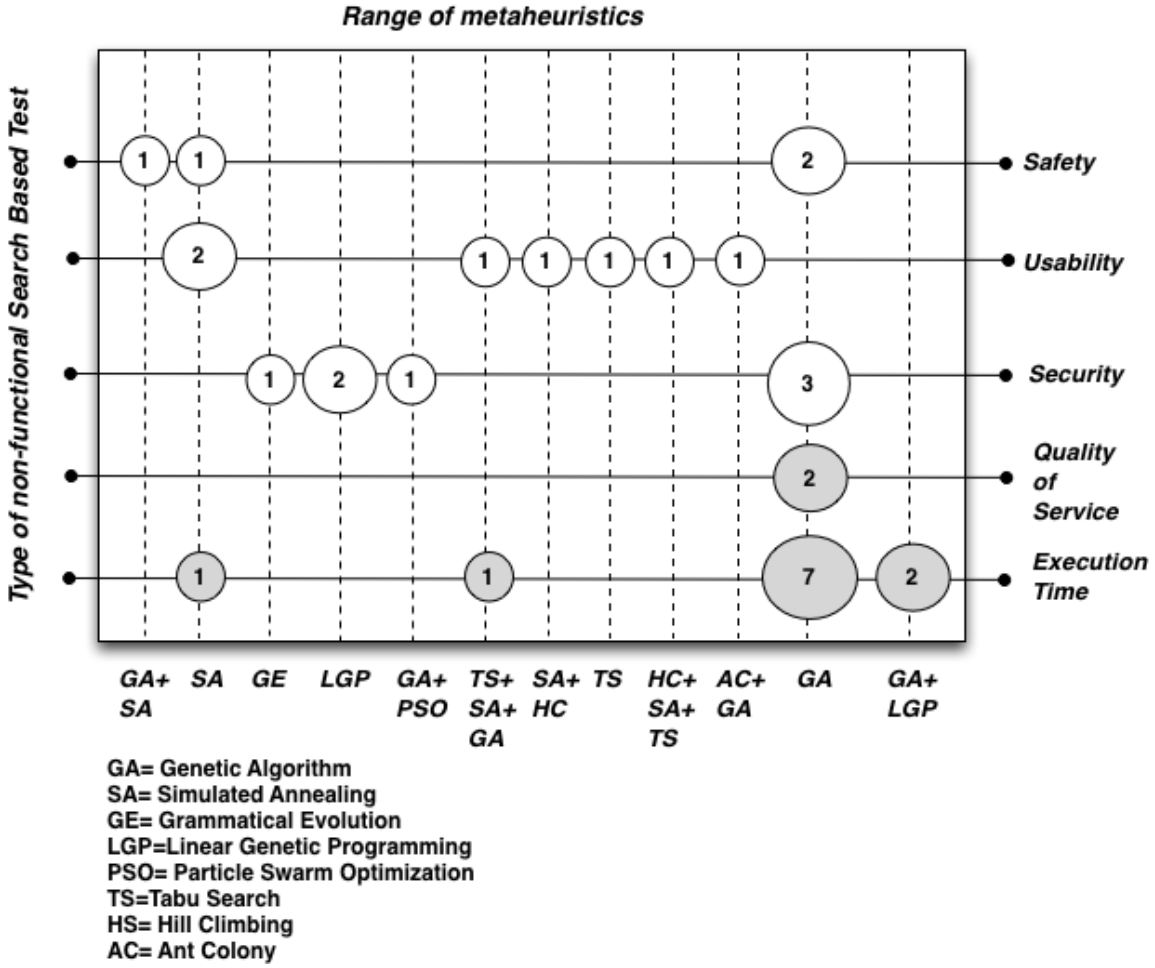


Figure 3-2: Range of metaheuristics by Type of non-functional Search Based Test[2].

3.4 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 [57]. 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 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] [61].

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 for each platform. Execution time measurement is a non deterministic approach, there is no guarantee to get the same results for 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 [45] [2].

Table 3.1 shows a comparison between the research studies on load, performance, and stress tests presented by Afzal et al. [2]. Afzal's work adds to some of the latest research in this area ([25] [27] [19] [20] [5] [31]). 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).

Wegener et al. [65] used genetic algorithms(GA) 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 [65].

Alander et al. [3] performed experiments in a simulator environment to measure response time extremes of protection relay software using genetic algorithms. The fitness function used was the response time of the tested software. The results showed that GA

Table 3.1: Distribution of the research studies over the range of applied metaheuristics

	Prototypes		**Functional Tool**
	Execution Time	Processor Cycles	Execution Time
GA + SA + Tabu Search			Gois et al. 2016 [31]
GA	Alander et al.,1998 [3] Wegener et al., 1996 and 1997 [65][38] Sullivan et al., 1998 [57] Briand et al., 2005 [13] Canfora et al., 2005 [15]	Wegener and Grochtmann, 1998 [64] Mueller et al., 1998 [46] Puschner et al. [50] Wegener et al., 2000 [66] Gro et al., 2000 [34]	Di Penta, 2007 [47] Garoussi, 2006 [25] Garousi, 2008 [26] Garousi, 2010 [27]
Simulated Annealing (SA)			Tracey, 1998 [60]
Constraint Programming			Alesio, 2014 [20] Alesio, 2013 [19]
GA + Constraint Programming			Alesio, 2015 [5]
Customized Algorithm		Pohlheim, 1999 [48]	

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 duration of execution 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 [38] [64] .

Gro et. al. [34] 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 [34].

Tracey et al. [60] used simulated annealing (SA) to test four simple programs. The results of the research presented that the use of SA was more effective with larger parameter space. The authors highlighted the need of a detailed comparison of various optimization techniques to explore WCET and BCET of the of the system under test [60].

Pohlheim and Wegener used an extension of genetic algorithms with multiple subpopulations, 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[48].

Briand et al. [13] 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 genetic algorithm. Two case studies were conducted and results illustrated that RTTT was a useful tool to stress a system under test [13].

Di Penta et al. [47] used GA to create test data that violated 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. The second case study, a service producing charts, applied the black-box approach with fitness calculated only on the basis of how close solutions violate QoS constraint. In case of audio workflow, the GA outperformed random search. For the second case study, use of black-box approach successfully violated the response time constraint, showing the violation of QoS constraints for a real service available on the Internet [47].

Garousi presented a stress test methodology aimed at increasing 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 [25].

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). 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 [5].

Gois et al. proposes an 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 significant 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 six generations previously established [31].

Chapter 4

Metaheuristics

In the computer science, the term metaheuristic is accepted for general techniques which are not specific to a particular problem. A metaheuristic is formally defined as an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space [51].

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 [11]:

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

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

- One vs. various neighborhood structures. Most metaheuristic algorithms work on one single neighborhood structure. In other words, the fitness landscape topology

does not change in the course of the algorithm. Other metaheuristics, such as Variable Neighborhood Search (VNS), use a set of neighborhood structures which gives the possibility to diversify the search by swapping between different fitness landscapes.

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

Simulated Annealing (SA) is a randomized algorithm that tries to avoid being trapped in local optimum solution by assigning probabilities to deteriorating moves. The SA procedure is inspired from the annealing process of solids. SA is based on a physical process in metallurgy discipline or solid matter physics. Annealing is the process of obtaining low energy states of a solid in heat treatment [39].

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 [51].

Algorithm 1 Simulated Annealing Algorithm

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

Tabu Search is a metaheuristic that guides a local heuristic search procedure to explore the solution space beyond local optimal and search with short term memory to avoid cycles.

Tabu Search uses a tabu list to keep track of the last moves, and don't allow going back to these [30].

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

Algorithm 2 Tabu Search Algorithm

$s \leftarrow GenerateInitialSolution()$
InitializeTabuLists(TL_1,...,TL_r)
while termination conditions not met **do**
  $N_a(s) \leftarrow \{s_1 \in N(s) | s_1$ does not violate a tabu condition, or it satisfies at least one aspiration condition $\}$
  $s_1 \leftarrow argmin\{f(s_2) | s_2 \in N_a(s)\}$
  UpdateTabuLists(TL_1,...,TL_r,s,s_1)
  $s \leftarrow s_1$
end while

Population-based metaheuristics (P-metaheuristics) could be viewed as an iterative improvement in a population of solutions. First, the population is initialized. Then, a new population of solutions is generated. Finally, this new population is integrated into the current one using some selection procedures. The search process is stopped when a stopping criterion is satisfied. Algorithms such as Genetic algorithms (GA), scatter search (SS), estimation of distribution algorithms (EDAs), particle swarm optimization (PSO), bee colony (BC), and artificial immune systems (AISs) belong to this class of metaheuristics [58].

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 [51].

Algorithm 3 Genetic Algorithm

```
    s ← GenerateInitialSolution()
    Evaluate(P)
3:  while termination conditions not met do
        P1 ← Recombine(P)
        P2 ← Mutate(P1)
6:      Evaluate(P2)
        P ← Select(P2, P)
    end while
```

4.1 Hybrid Metaheuristics

However, in recent years it has become evident that the concentration on a sole metaheuristic is rather restrictive. A skilled combination of a metaheuristic with other optimization techniques, a so called hybrid metaheuristic, can provide a more efficient behavior and a higher flexibility when dealing with real-world and large-scale problems [59].

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 [51].

There are two main categories of metaheuristic combinations: collaborative combinations and integrative combinations. These are presented in Fig. 4-1 [52].

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 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 success-

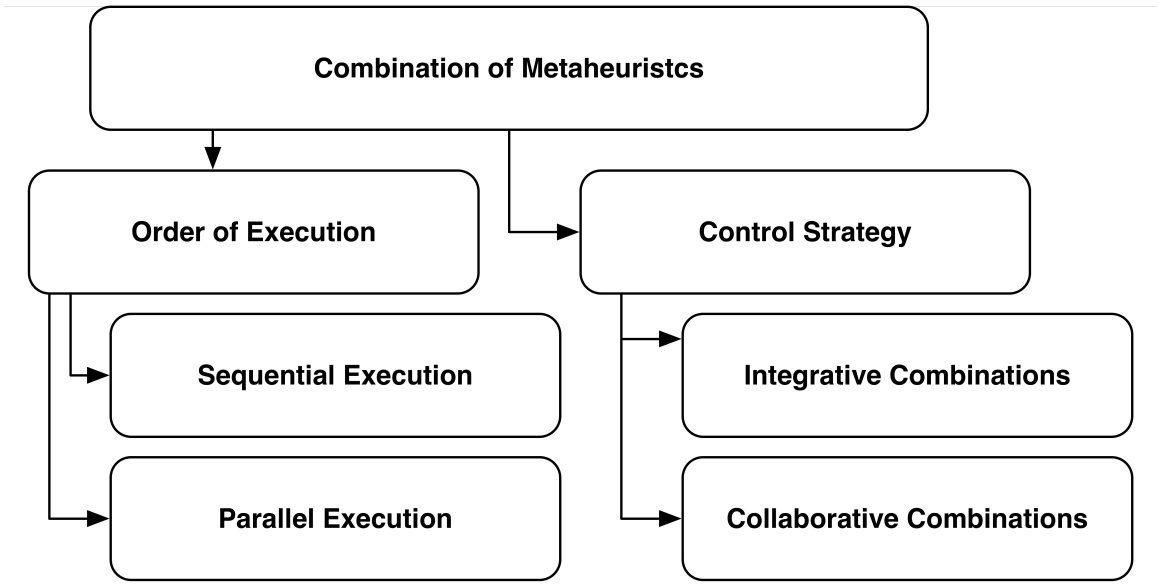


Figure 4-1: Categories of metaheuristc combinations [49]

ful [51].

The work uses a type of collaborative combination with sequential execution with two trajectory methods (Tabu Search and Simulated Annealing) and Genetic Algorithms.

Chapter 5

Q-Learning

Chapter 6

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

6.1 Hybrid Approach

A large number of researchers have recognized the advantages and huge potential of building hybrid metaheuristics. The main motivation for creating hybrid metaheuristics is to exploit the complementary character of different optimization strategies. In fact, choosing an adequate combination of algorithms can be the key to achieving top performance in solving many hard optimization problems [49] [12].

The proposed solution makes it possible to create a model that evolves during the test. 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 in others 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. 6-1).

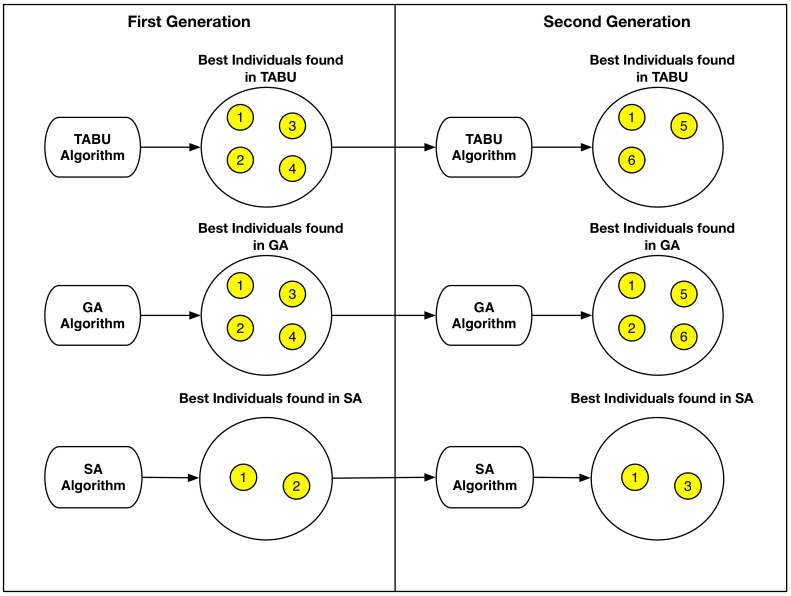


Figure 6-1: Use of the algorithms independently

The second approach uses the algorithms in a collaborative mode (hybrid metaheuristic). In this approach, the three algorithms share their best individuals found (Fig. 6-2). The next subsections present details about the used metaheuristic algorithms (Representation, initial population and fitness function).

6.1.1 Representation

The solution representation is composed by a linear vector with 23 positions. The first position represents the name of an individual. The second position represents the algorithm (genetic algorithm, simulated annealing, or Tabu search) used by the individual. The third position represents the type of test (load, stress, or performance). The next positions represent 10 scenarios and their numbers of users. Each 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.

Fig. 6-3 presents the solution representation and an example using the crossover operation. In the example, genotype 1 has the Login scenario with 2 users, the Form scenario with 0 users, and the Search scenario with 3 users. Genotype 2 has the Delete scenario with 10 users, the Search scenario with 0 users, and the Include scenario with 5 users. After the

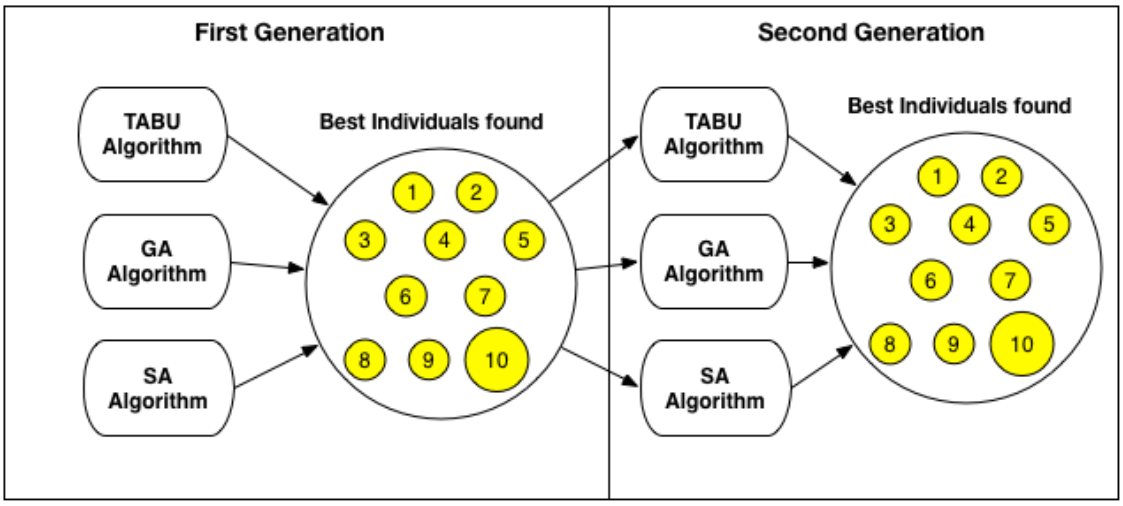


Figure 6-2: Use of the algorithms collaboratively

crossover operation, we obtain a genotype with the Login scenario with 2 users, the Search scenario with 0 users, and the Include scenario with 5 users.

Fig. 6-4 shows the strategy used by the proposed solution to obtain the representation of 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.

6.1.2 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.

The percentages relates 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.

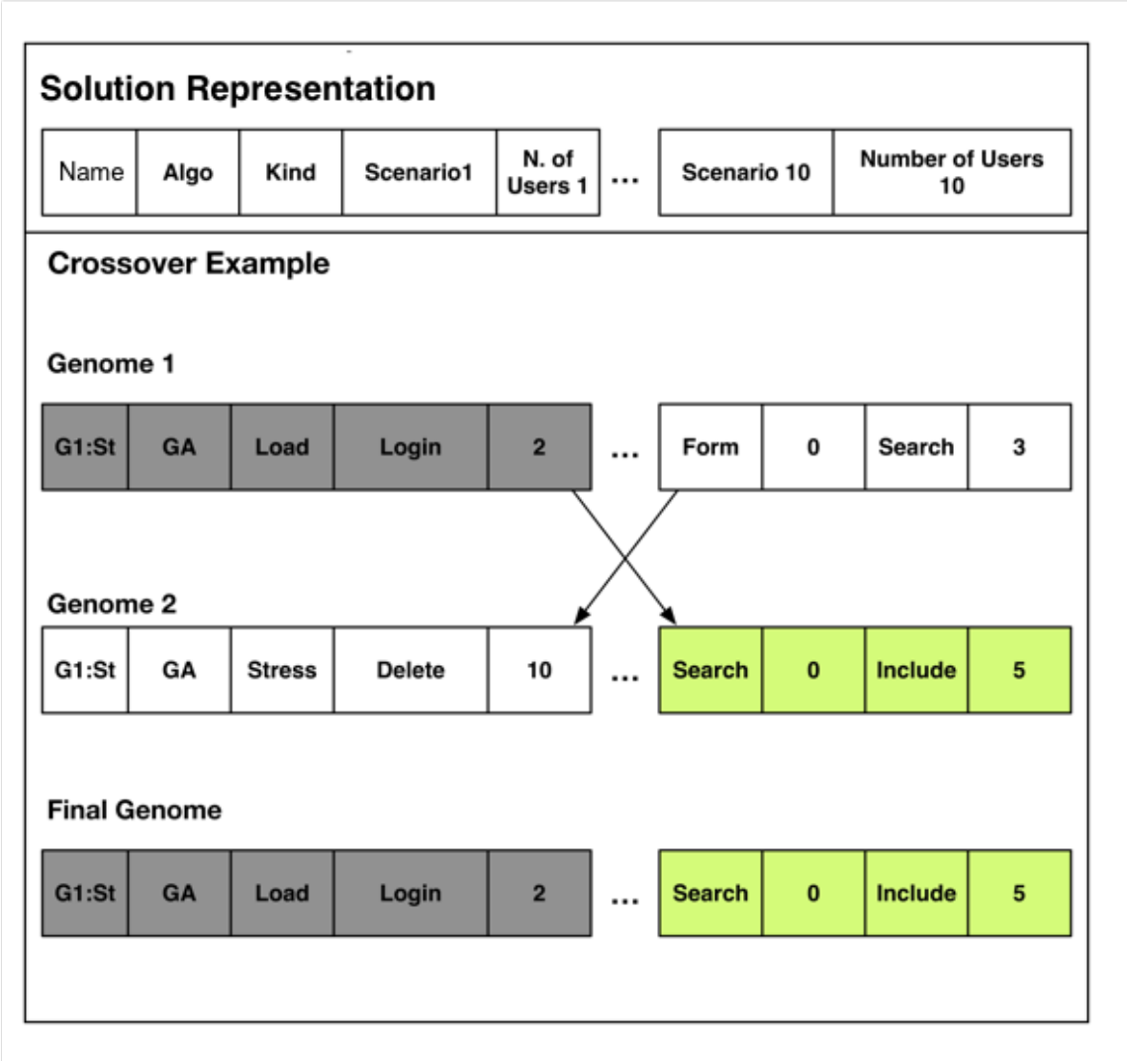


Figure 6-3: Solution representation and crossover example

6.1.3 Objective (fitness) function

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

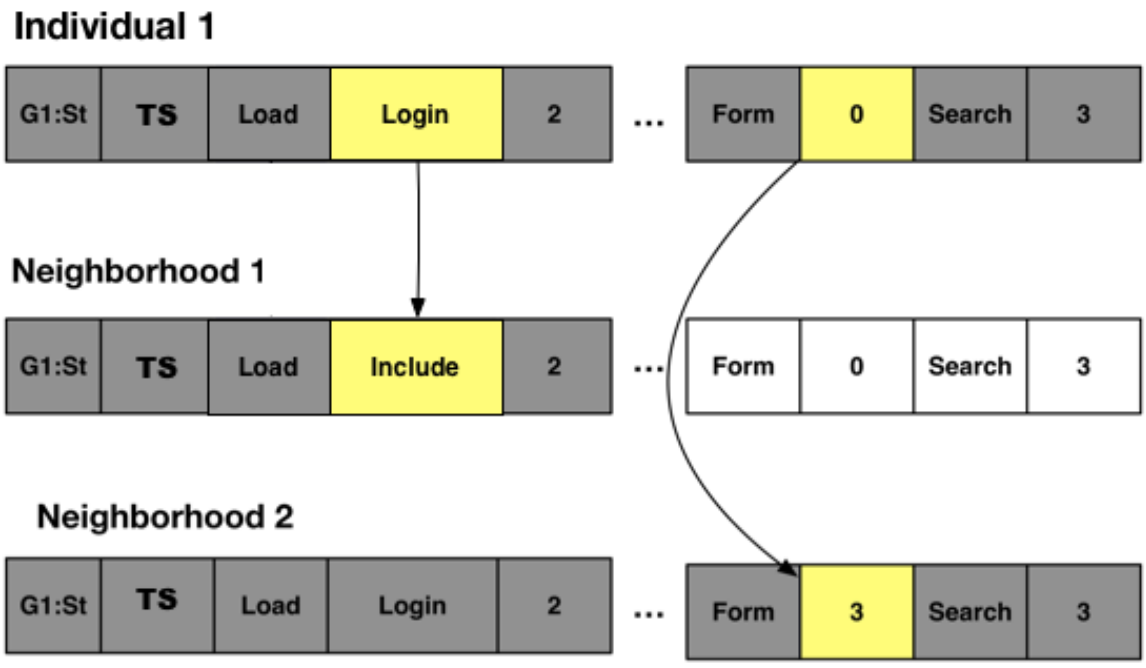


Figure 6-4: Tabu search and simulated annealing neighbor strategy

$$\begin{aligned} fit = {} & 90percentileweigth * 90percentiletime \\ & +80percentileweigth * 80percentiletime \\ & +70percentileweigth * 70percentiletime+ \\ & maxResponseWeigth * maxResponseTime+ \\ & numberOfUsersWeigth * numberOfUsers - penalty \end{aligned} \tag{6.1}$$

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 an application under test takes a longer time to respond than the level of service. The penalty is calculated by the follow equation:

$$\begin{aligned} penalty &= 100 * \Delta \\ \Delta &= (t_{CurrentResponseTime} - t_{MaximumResponseTimeExpected}) \end{aligned} \tag{6.2}$$

6.2 IAdapter

IAdapter is a JMeter plugin designed to perform search-based stress tests. The plugin is available on www.iadapter.org. The IAdapter plugin implements the solution proposed in Section 5. The next subsections present details about the Apache JMeter tool, the IAdapter Life Cycle and the IAdapter Components. The IAdapter plugin provides three main components: WorkLoadThreadGroup, WorkLoadSaver, and WorkLoadController.

The Fig. 6-5 show the IAdapter architecture. All metaheuristic class implements the interface IAlgorithm. Test scenarios and test results are stored in a Mysql database. GeneticAlgorithm class uses a framework named JGAP to implement Genetic Algorithms.

The WorkLoadThreadGroup class is the Load Injection and Test Management modules, responsible to generate the initial population and uses the JMeter Engine to realize requests to server under test.

6.2.1 IAdapter Life Cycle

Fig. 6-6 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.

6.2.2 IAdapter Components

WorkLoadThreadGroup is a component that creates an initial population and configures the algorithms used in IAdapter. Fig. 6-7 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.

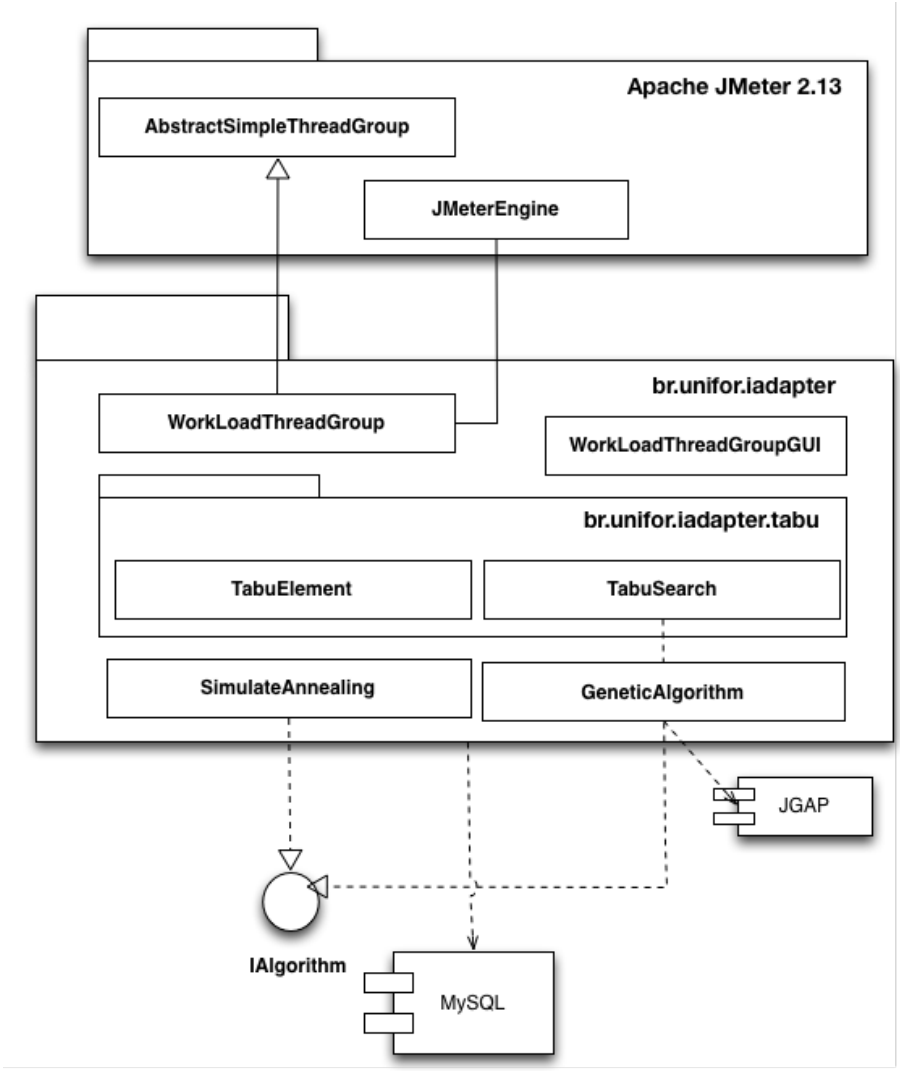


Figure 6-5: IAdapter architecture

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 need to login into the application under test and bring the application back to its original state.

6.2.3 IAdapter Testbed Tool

A Testbed makes possible follow a formalized methodology and reproduce tests for further analysis and comparison. It seems natural that one of the most important parts of a comparison among heuristics is the testbed on which the heuristics are tested. As a result, the testbed should be the first consideration when comparing two metaheuristics [29].

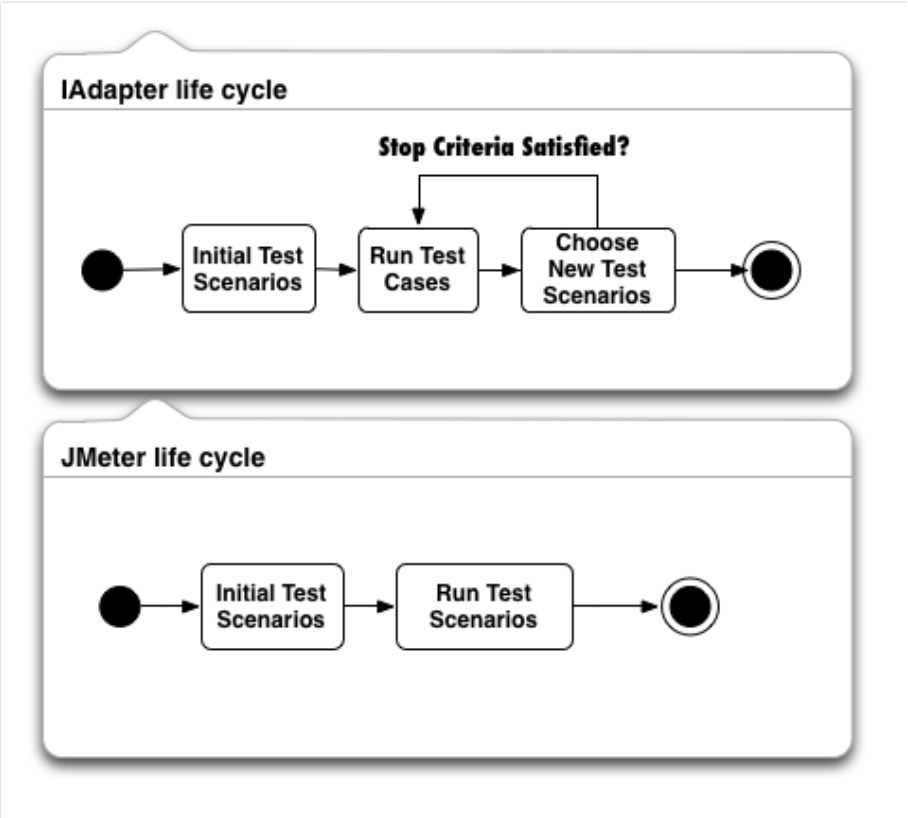


Figure 6-6: IAdapter life cycle

In this section, We devise a new testbed that has the ability to reproduce different types of web workloads. The proposed solution extends the IAdapter plugin to create a testbed tool to validade load, performance and stress search based tests approaches [31]. The IAdapter is a JMeter plugin designed to perform search-based stress tests.

This new testbed must accomplish three main goals. First, it must reproduce a workload by using an antipattern implementation. Second, it must be able to provide client and server metrics with the aim of being used for web performance evaluation studies. Finally, it is should be extensible, allowing create new test scenarios.

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 testbed tool proposed consists of four main elements. The first element is a emulator module that it is responsible to simulate the antipatterns in a specific context. The

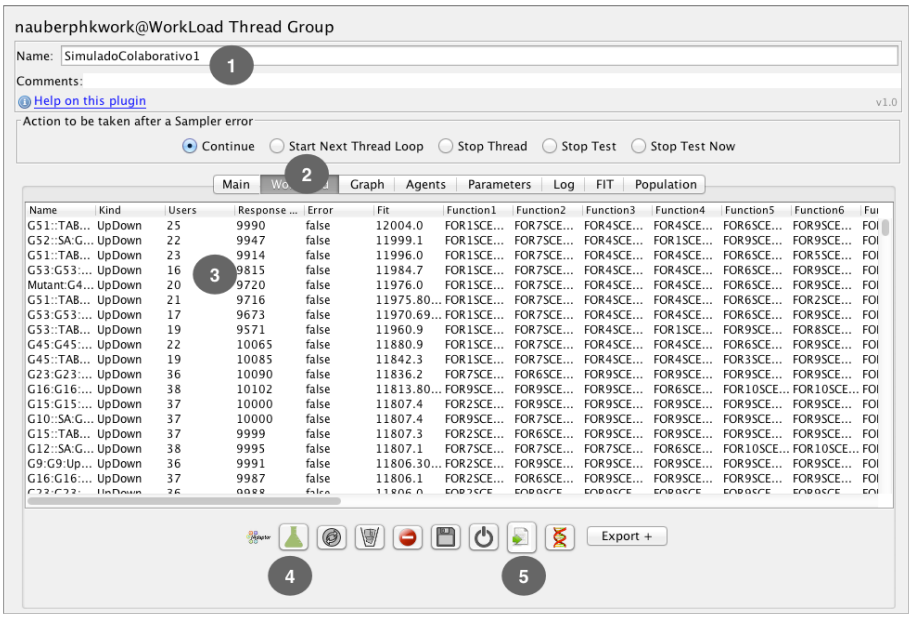


Figure 6-7: WorkLoadThreadGroup component

second it's a module named test module that it is responsible for use a previous selected metaheuristics and perform a search based test. The third module contains the test scenarios representation. The fourth module it is responsible for provide a service of explore the neighborhood of a given individual.

Testbed Architecture

The Fig. 6-8 presents the main architecture of the Testbed 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 tests for each metaheuristic found. The Test Scenario Representation and Persistent Module provides the scenario representation used by the metaheuristics and persist the testbed results data in a database. Neighborhood provider service is responsible to search neighbors of some individual provided as parameter to the service.

Test Module

The Test Module is responsible for load all classes that extends AbstractAlgorithm in the classpath and perform the tests under the application. The Emulator Module provides suc-

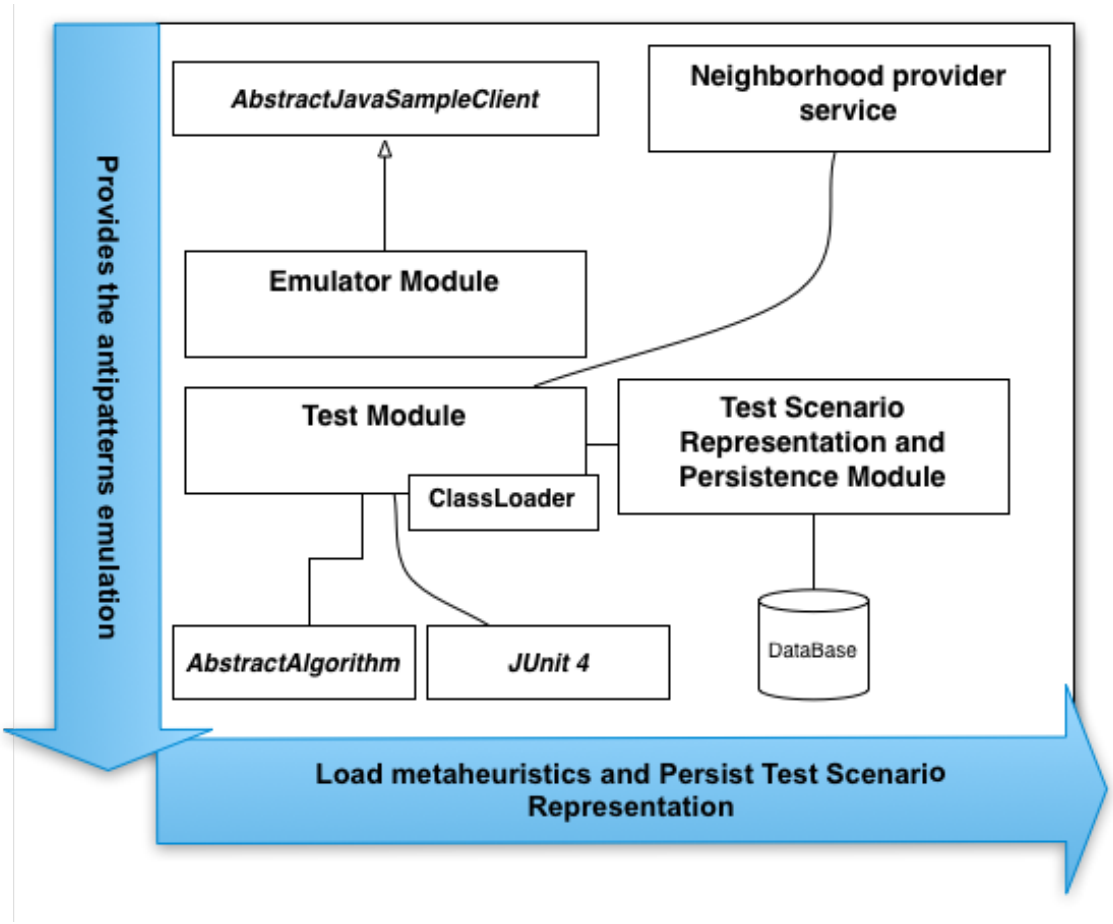


Figure 6-8: testbed main architecture.

cessful scenarios and antipatterns implementations. The heuristics are executed in order to select the scenarios with failures or high response times. The Fig. 6-10 presents the first feature of Test Module where a initial population it is created and IAdapter with JMeterEngine performs all tests and apply a fitness value to each workload.

The Fig. 6-11 presents the Test Module life cycle. The life cycle iterate over two steps: The first step apply a metaheurist to select or generate a new set of workloads based on selection criteria. The second step run each workload with the JMeterEngine and obtain a fitness value based on some objective function. The red circles represent the workload that contain errors. The green circles represents the workloads with no errors and low acceptable response time. The testbed tool uses as default objective function the equation:

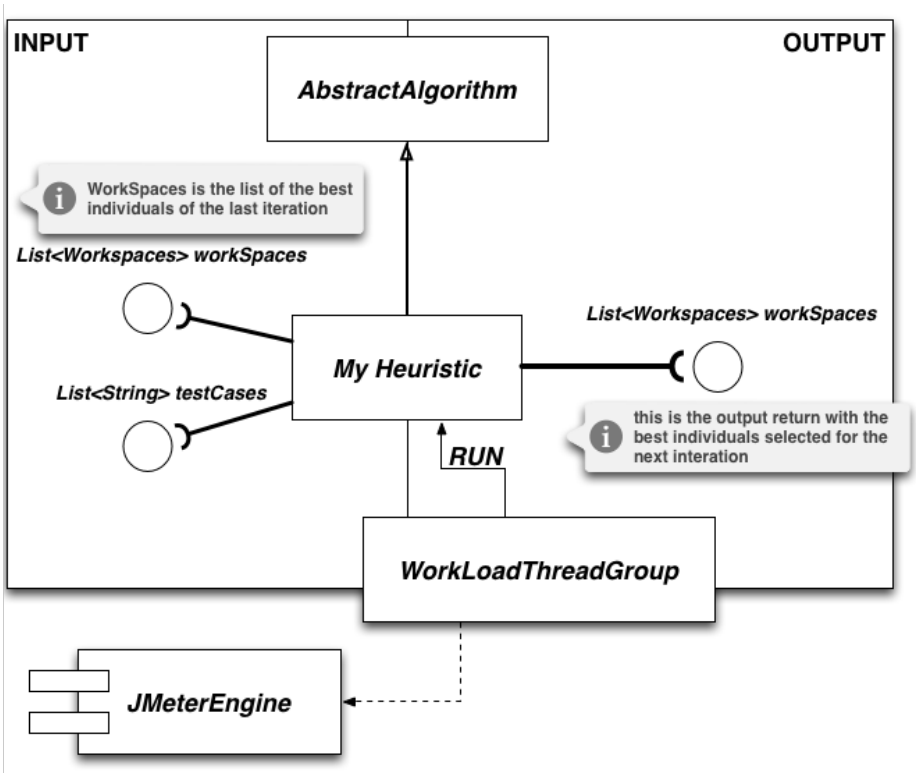


Figure 6-9: Heuristic class diagram.

$$\begin{aligned} fitness = {} & 90percentileweigth * 90percentiletime \\ & +80percentileweigth * 80percentiletime \\ & +70percentileweigth * 70percentiletime+ \\ & maxResponseWeigth * maxResponseTime+ \\ & numberOfUsersWeigth * numberOfUsers - penalty \end{aligned} \tag{6.3}$$

The use of presented fitness value by each metaheuristic it's optional. Each Metaheuristic could define your own objective function. The proposed 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 an application under test takes a longer time to respond than the level of service. After all these steps the cycle begins until the maximum number of generations it is reached. The Fig. 6-9 shows the class diagram for custom and provided heuristics. All heuristic classes extends

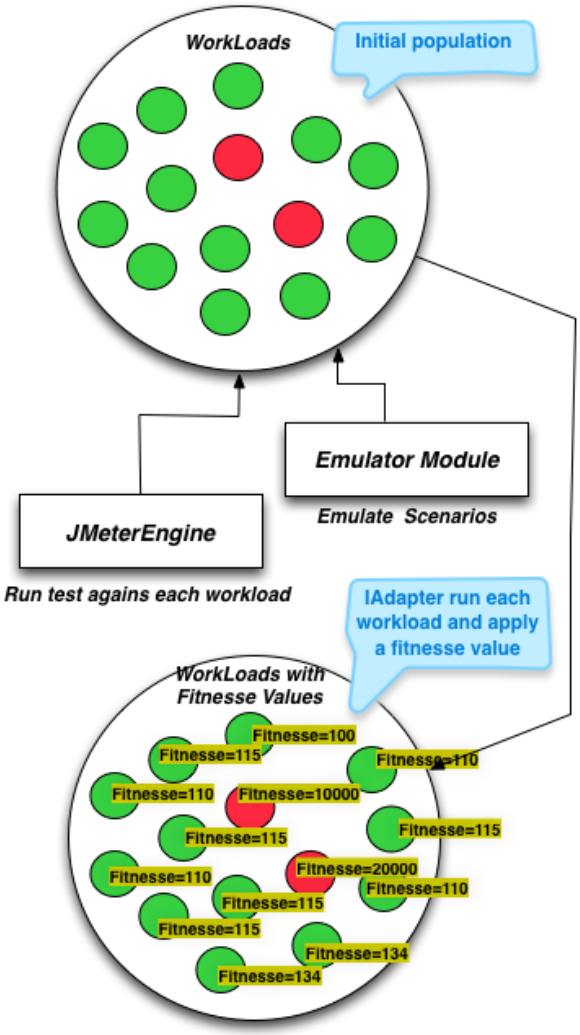


Figure 6-10: Test Module first feature.

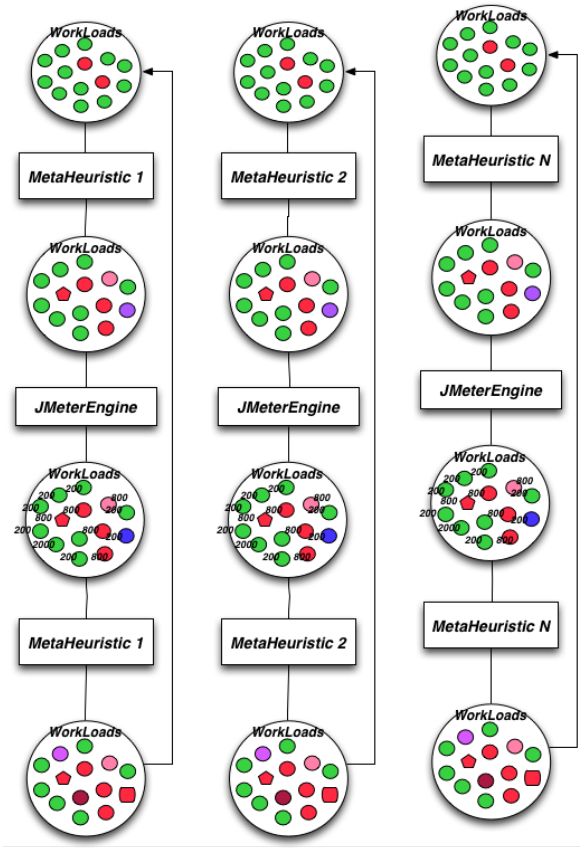


Figure 6-11: Test Module life cycle.

the class AbstractAlgorithm. The heuristics receives as input a list of workspaces and a list of testcases. The workspace represents each individual in the search space.

Each metaheuristic class returns a list of workspaces (the individuals selected to the next generation). The Listing presents the method that performs the search of classes that extends Abstract Algorithm

Emulator Module

The Emulator Module is responsible for implement and provide successful scenarios and the most commons performance antipatterns. All classes must extends the AbstractJavaSamplerClient class or use JUnit 4. The AbstractJavaSamplerClient class allows create a JMeter

Java Request. Using JUnit 4, the emulators classes could be called by a JMeter JUnit request. The Fig. 6-12 presents the main features of the emulator module. The module implements 8 test scenarios in its first version.

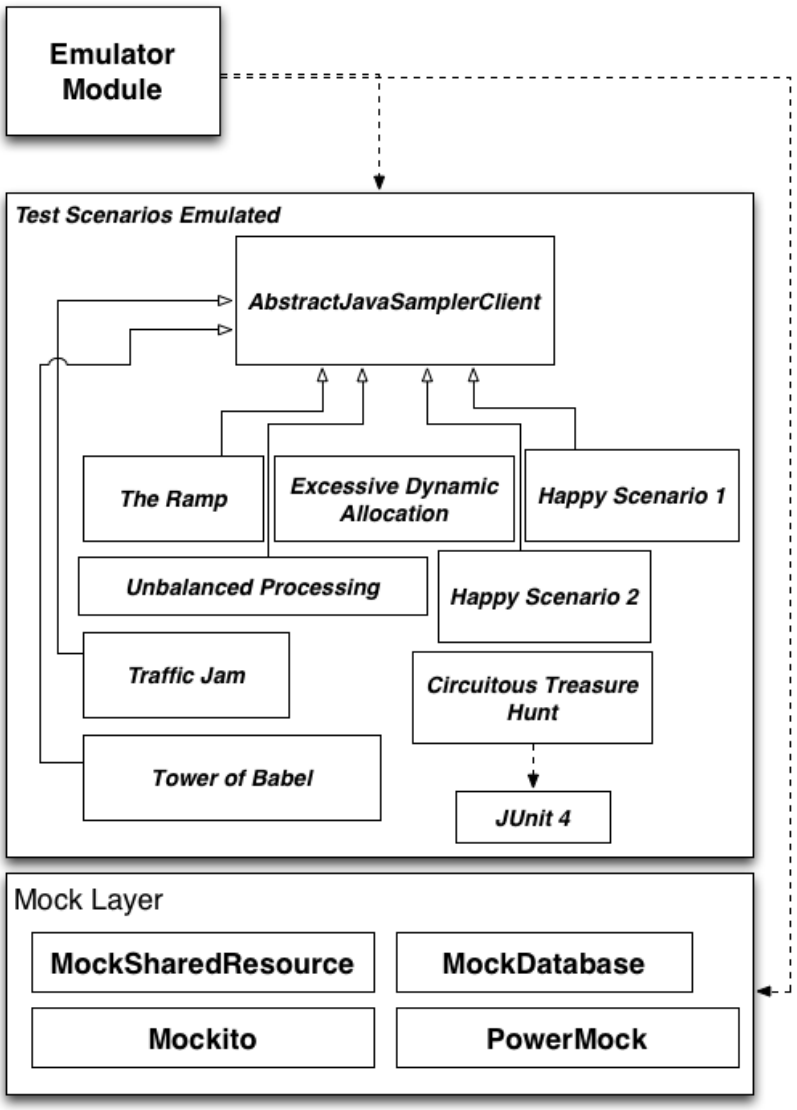


Figure 6-12: Heuristic class diagram.

The Mock Layer provides emulated databases and components to the test scenarios. Each scenario provided by the Emulator Module could be called in JMeter using a Java Request. The algorithm 4 emulates the Unbalanced Processing antipattern. The test scenario C still waiting until A and B scenarios are used by a test.

Algorithm 4 Unbalanced Processing emulate algorithm

```
1: while List of processing scenarios contains A and B do
2:     Processing A and B scenarios
3: end while
4: Processing scenario C
```

The algorithm 5 emulates the Ramp antipattern. The algorithm increase the response time at each time that it has been used.

Algorithm 5 The Ramp emulate algorithm

```
1: if count is null then
2:     count ← 0
3: end if
4: sleep(100* count)
5: count ← count + 1
```

The algorithm 6 implements the Excessive Dynamic Allocation antipattern. The algorithm creates a connection with a emulated database, uses the connection and finally the connection.

Algorithm 6 Excessive Dynamic Allocation emulate algorithm

```
1: for each request do
2:     for int i=0 to 1000 do
3:         Create a connection to a database
4:         Use the connection
5:         Destroy the created connection
6:     end for
7: end for
```

The algorithm 7 presents the Happy Scenario 1. The response time increases for every 10 users.

Algorithm 7 Happy Scenario 1 emulate algorithm

```
1: sleep(2*users)
```

Algorithm 8 Happy Scenario 2 emulate algorithm

```
1: sleep(3*users)
```

A further 4 algorithms were developed for the scenarios Circuitous Treasure Hunt, Happy Scenario 2, Traffic Jam and Tower of Babel.

Chapter 7

Experiments

In this chapter, We present the results of experiments which we carried out to verify the hybrid and q-learning algorithm approaches ,the antipatterns implementation, the fitness objective function and the metaheuristics used by the testbed tool.

7.0.1 Emulated Class Test Experiment

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.

The experiment used the following fitness function:

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

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:   }
```

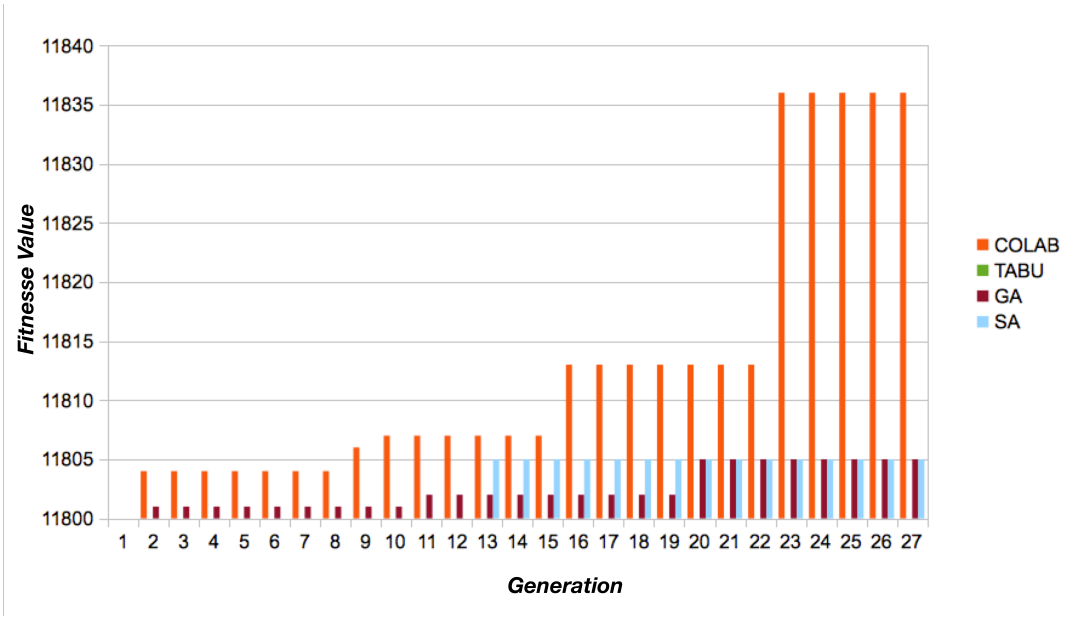
This fitness function is the same function represented in the section VII with the manually adjustable user-defined weights filled out. This fitness function intended to find individuals with the highest percentile of 90%, followed by individuals with a higher percentile time of 80% and 70%, maximum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

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,

Figure 7-1: Best results obtained in 27 generations



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.

7.1 Testbed Tool Experiments

We conducted two experiments in order to verify the effectiveness of the testbed tool. The experiments ran for 17 generations. The experiments used an initial population of 4 in-

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

dividuals by metaheuristic. 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. The experiments uses tabu search, genetic algorithms and the hybrid metaheuristic approach proposed by Gois et al. [31].

The objective function applied is intended to maximize the number of users and minimize the response time of the scenarios being tested. In this experiments, better fitness values meaning to find scenarios with more users and a low values of response time. A penalty is applied when the response time is greater than the maximum response time ex-

pected. The experiments used the following fitness (goal) function. :

$$\begin{aligned} fit = 3000 * numberOfUsers \\ -20 * 90percentiletime \\ -20 * 80percentiletime \\ -20 * 70percentiletime \\ -20 * maxResponseTime \\ -penalty \end{aligned} \tag{7.2}$$

The experiments addresses:

- Validate the operation of the testbed tool.

- Find the maximum number of users and the minimal response time.

- Analyze and verify the best heuristics among those chosen to the experiments.

7.1.1 The Ramp and Circuitous Treasure Hunt experiment

The experiment was carried out for 8 continuous hours. All tests in the experiment were conducted without the need of a tester, automating the process of executing and designing performance test scenarios.In this experiment, Scenarios were generated with the Ramp and Circuitous Treasure antipattern as well as scenarios with Happy Scenario 1, Happy Scenario 2 and mixed scenarios. The Fig. 7-2 and 7-3 presents the fitness value obtained by each metaheuristic. The SA algorithm obtained the worst fitness values. Hybrid metaheuristic obtained the better fitness values.

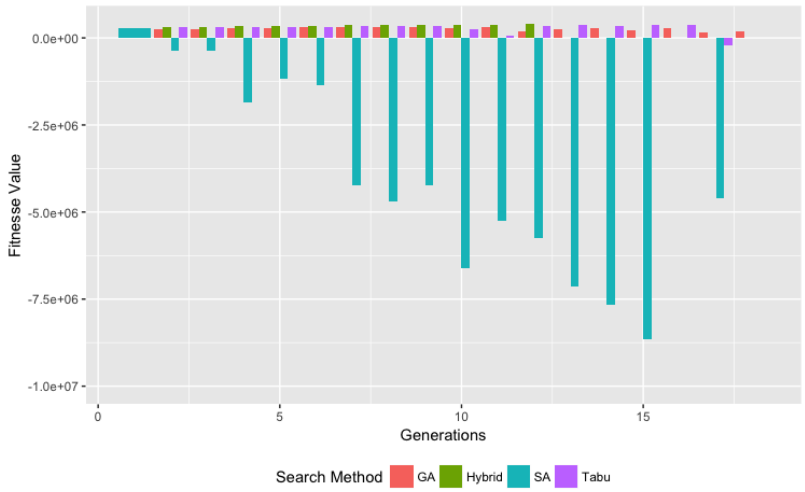


Figure 7-2: fitness value obtained by Search Method

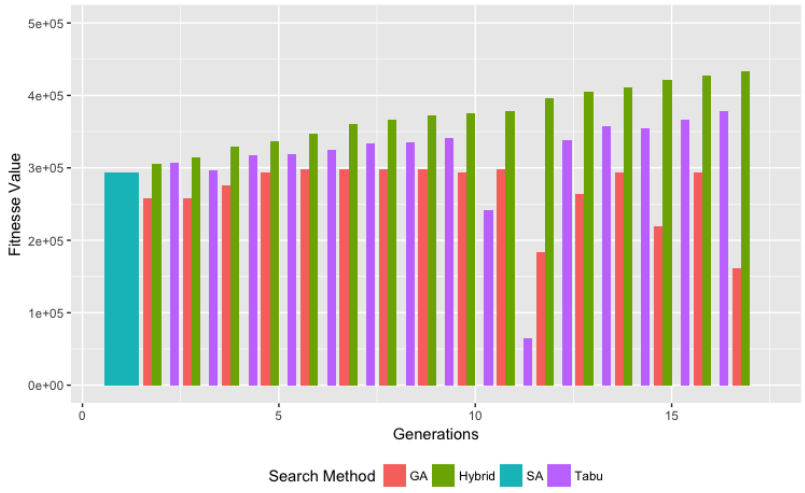


Figure 7-3: fitness value obtained by Search Method without SA metaheuristic.

Despite having obtained the best fitness value in each generation, the Hybrid algorithm performs twice as many requests as the second one, the tabu search (Fig. 7-4). The Fig. 7-5 shows the average, minimal e maximum value by search method.

Figure 7-4: Number of requests by Search Method

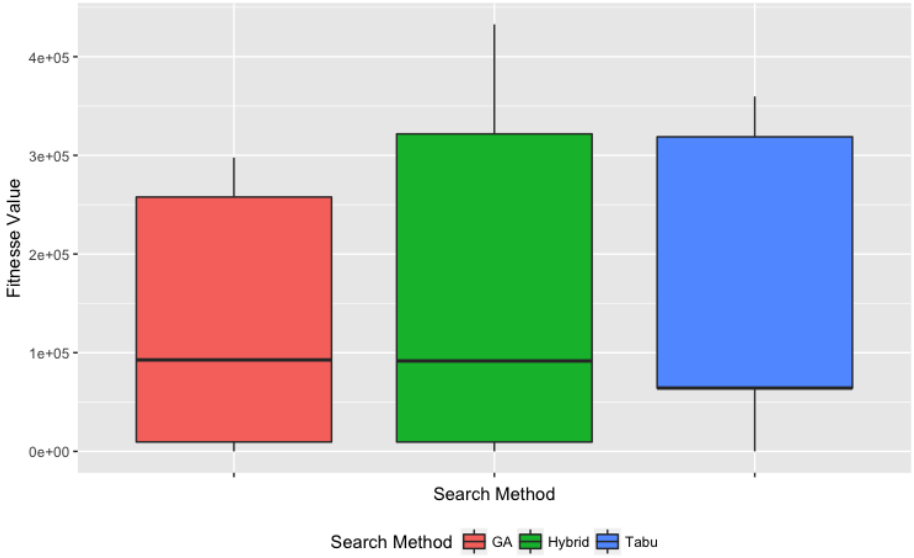


Figure 7-5: Average, median, maximum and minimal fitness value by Search Method

The Fig. 7-6 presents the maximum, average, median and minimum fitness value by generation. The maximun fitness value increases at each generation. The Fig. 7-7 presents the density graph of number of users by fitness value. The range between 100 and 150 users has the highest number of individuals found with higher fitness value.

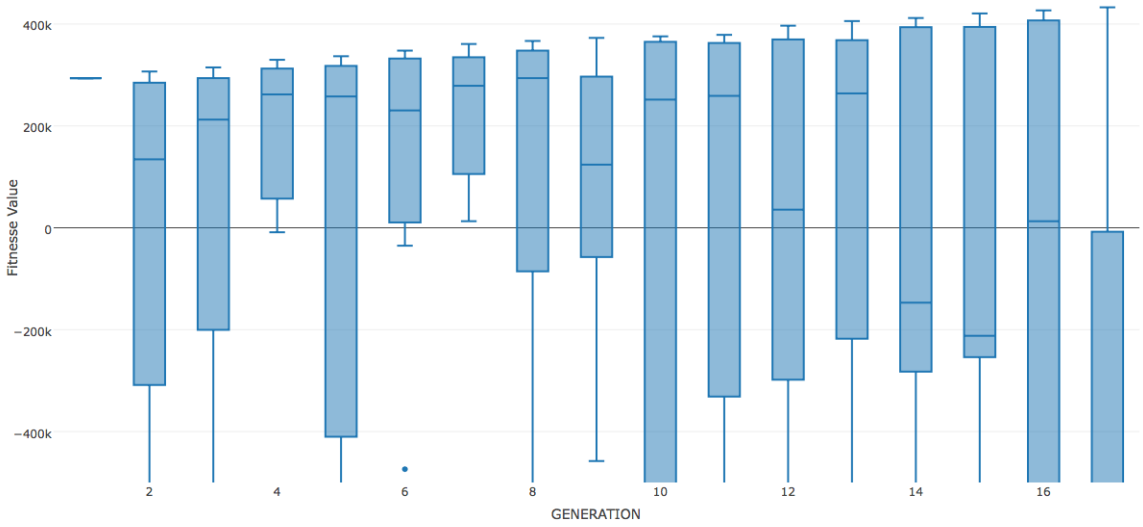


Figure 7-6: fitness value by generation

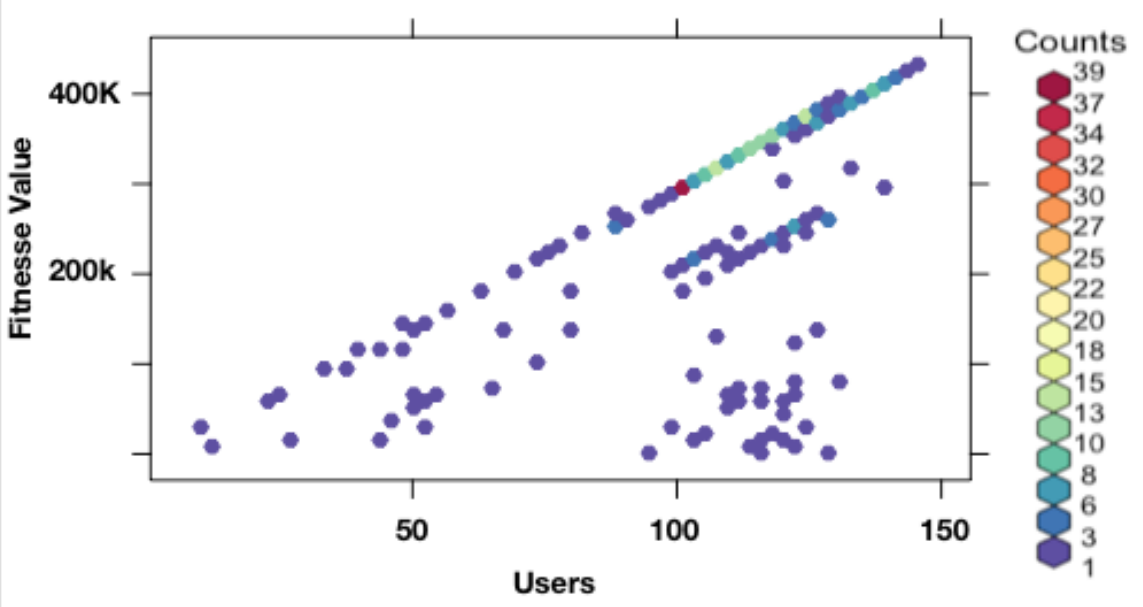


Figure 7-7: Density graph of number of users by fitness value

Table 7.2 shows 4 individuals with 143 to 146 users. These are the scenarios with the maximum number of users found with the best response time. The first individual has 64 users on Happy Scenario 2, 81 users on Happy Scenario 1 and a response time of 12 seconds. None of the best individuals has one of the antipatterns used in the experiment.

Fig. 7-8 presents the response time by number of users of individuals with Happy

Table 7.2: Best individuals found in the first experiment

Search Method	Generation	Users	fitness Value	Happy 2	Happy 1	Response Time
Hybrid	17	145	432760	64	81	12
Hybrid	17	145	432740	46	99	13
Hybrid	17	146	431760	54	92	12
Hybrid	16	143	426740	30	113	13

Scenario 1 and Happy Scenario 2. The Figure illustrates that the individuals with best fitness value has more users and minor response time. The Fig. 7-9 presents the response time by number of users of individuals with the Ramp and Circuitous Treasure antipatterns scenarios. The Figure illustrates the smallest number of individuals with the antipatterns when compared to individuals who use the happy scenarios.

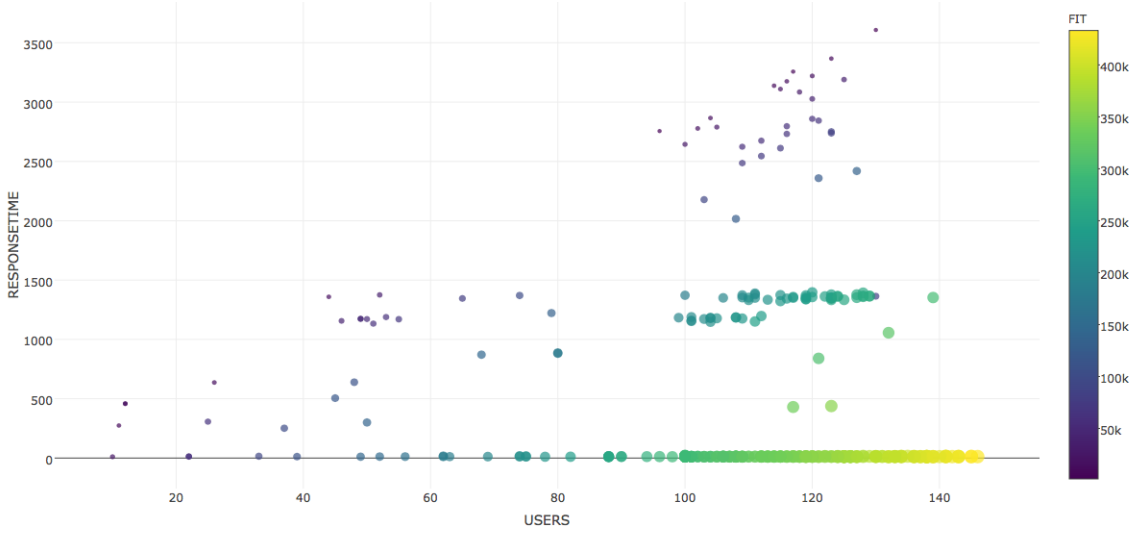


Figure 7-8: Response time by number of users of individuals with Happy Scenario 1 and Happy Scenario 2

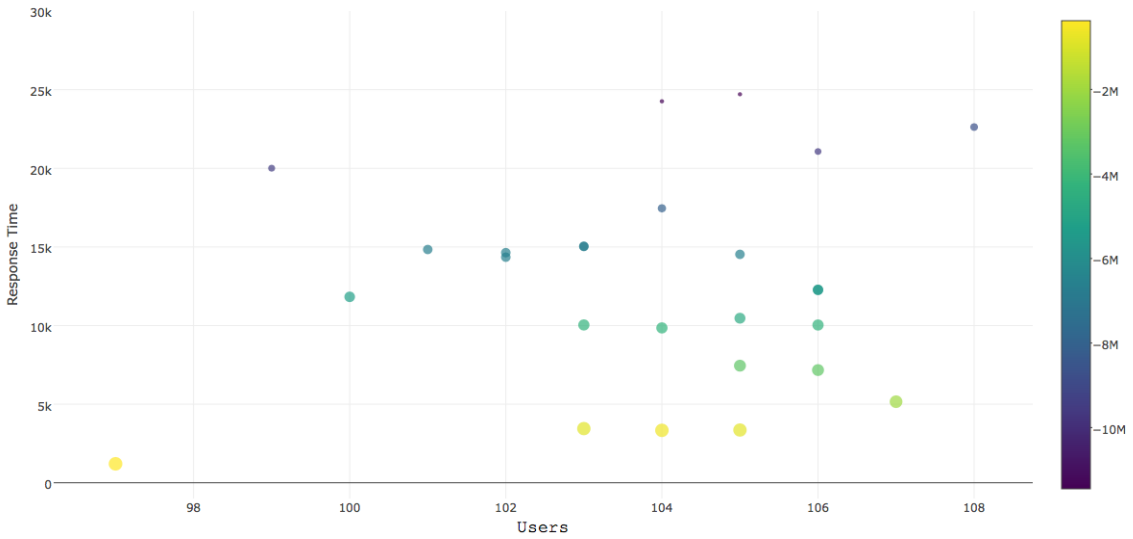


Figure 7-9: Response time by number of users of individuals with the Ramp and Circuitous Treasure antipatterns

In the first experiment, We conclude that the metaheuristics converged to scenarios with an happy path, excluding the scenarios with antipatterns. The hybrid 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 the Ramp and Circuitous Treasure antipatterns and found neighbors that still using the antipatterns over the 17 generations of the experiment.

7.1.2 The Tower Babel and Unbalanced Processing experiment

The experiment was carried out for 6 continuous hours. All tests in the experiment were conducted without the need of a tester. In this experiment, Scenarios were generated with Tower Babel and Unbalanced Processing antipattern as well as scenarios with Happy Scenario 1, Happy Scenario 2 and mixed scenarios. The Fig. 7-10 presents the fitness value obtained by each metaheuristic. The SA algorithm obtained the worst fitness values. Hybrid metaheuristic obtained the better fitness values.

As in the first experiment, the Hybrid algorithm performs twice as many requests as the second one, the tabu search (Fig. 7-11). The Fig. 7-12 shows the average, minimal e

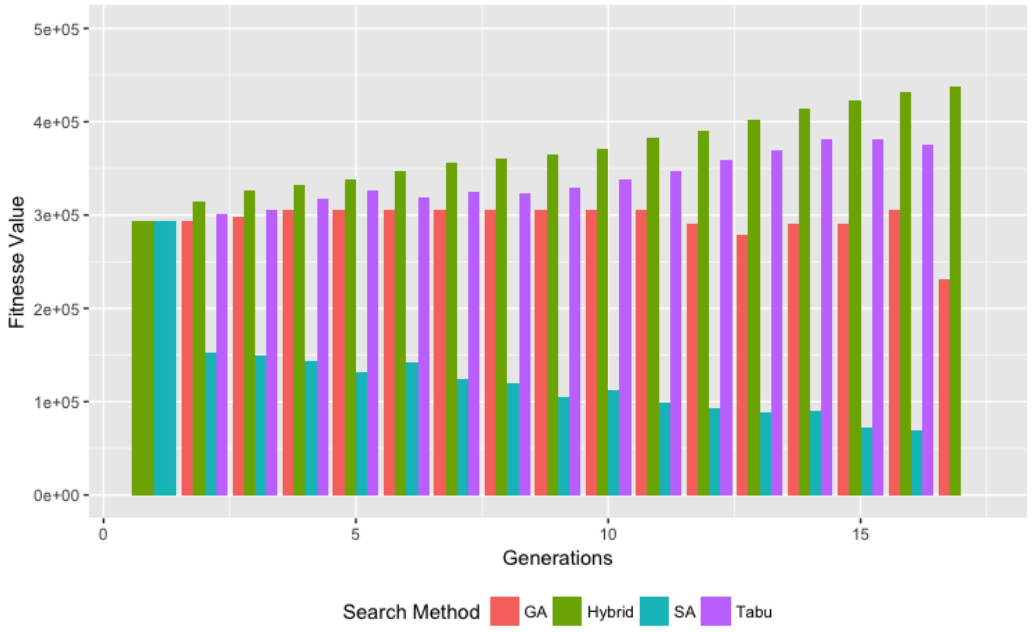


Figure 7-10: itnesse value obtained by Search Method

maximum value by search method. The Fig. 7-13 presents the maximum, average, median and minimum fitness value by generation. The maximun fitness value increases at each generation. The Fig. 7-14 presents the density graph of number of users by fitness value. The range between 100 and 150 users has the highest number of individuals found with higher fitness value.

Figure 7-11: Number of requests by Search Method

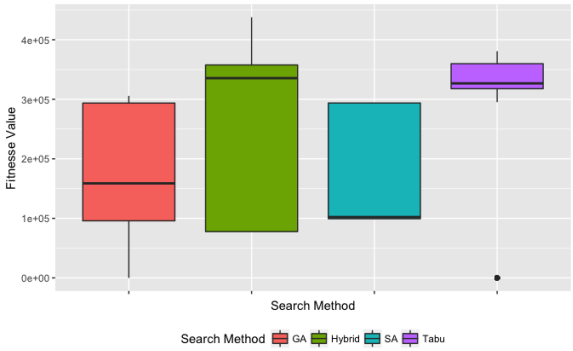


Figure 7-12: Finesse value by generation in all tests

Table 7.3 shows 4 individuals with 145 to 148 users. The first individual has 72 users on Happy Scenario 2, 30 users on Happy Scenario 1, 46 user with the antipattern Tower Babel and a response time of 11 seconds. Despite the fact of doing 300 conversions of the

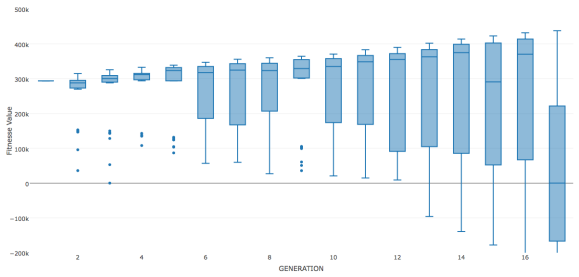


Figure 7-13: Response time by generation in all tests scenarios

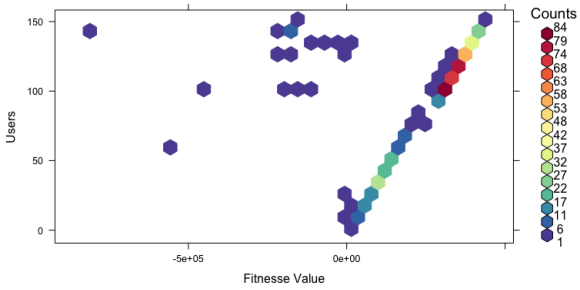


Figure 7-14: Finesse value by generation in all tests

JSON standard for XML. The antipattern implementation does not return a much higher response time than happy paths. While happy paths returns from 10 to 15 seconds from a single user, Tower Babel antipattern has a response time of 10 to 29 seconds. None of the best individuals found implements the Unbalanced Processing antipattern.

Fig. 7-15 presents the response time by number of users of individuals with Happy Scenario 1 and Happy Scenario 2. The Figure illustrates that the individuals with best fitness value has more users and minor response time. The Fig. 7-16 presents the response time by number of users of individuals with with Unbalanced Processing antipatterns scenarios. The Figure illustrates the smallest number of individuals with the Unbalanced Processing antipattern when compared to individuals who use the happy scenarios and the Tower Babel antipattern.

We conclude that the metaheuristics converged to scenarios with an happy path and Tower Babel antipattern, excluding the scenarios with Unbalanced Processing antipattern. The hybrid 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 an antipattern over the 17 generations of the experiment. The individual with best fitness value has 72 users on Happy Scenario 2, 30 users on Happy Scenario 1, 46 user with the antipattern Tower Babel and a response time of 11 seconds.

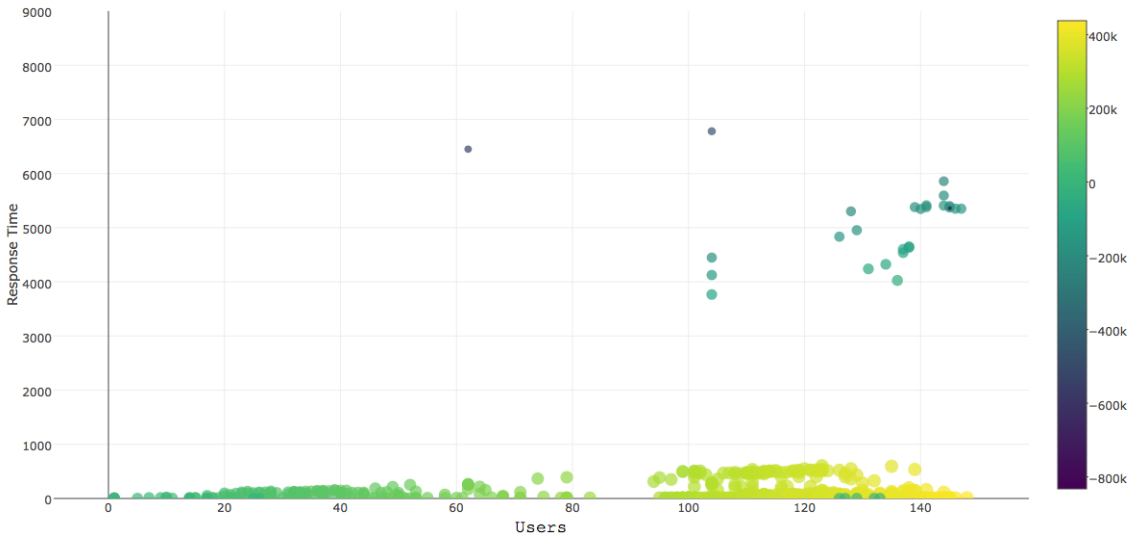


Figure 7-15: Response time by number of users of individuals with Happy Scenario 1, Happy Scenario 2 and Tower Babel antipattern

Table 7.3: Best individuals found in the second experiment

Search Method	Generation	Users	fitness Value	Happy 2	Tower	Happy 1	Time
Hybrid	17	148	437780	72	46	30	11
Hybrid	17	145	432740	71	15	59	13
Hybrid	16	146	431800	72	31	43	10
Hybrid	17	145	428780	71	32	42	11

7.1.3 Moodle Application Experiment

This 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.

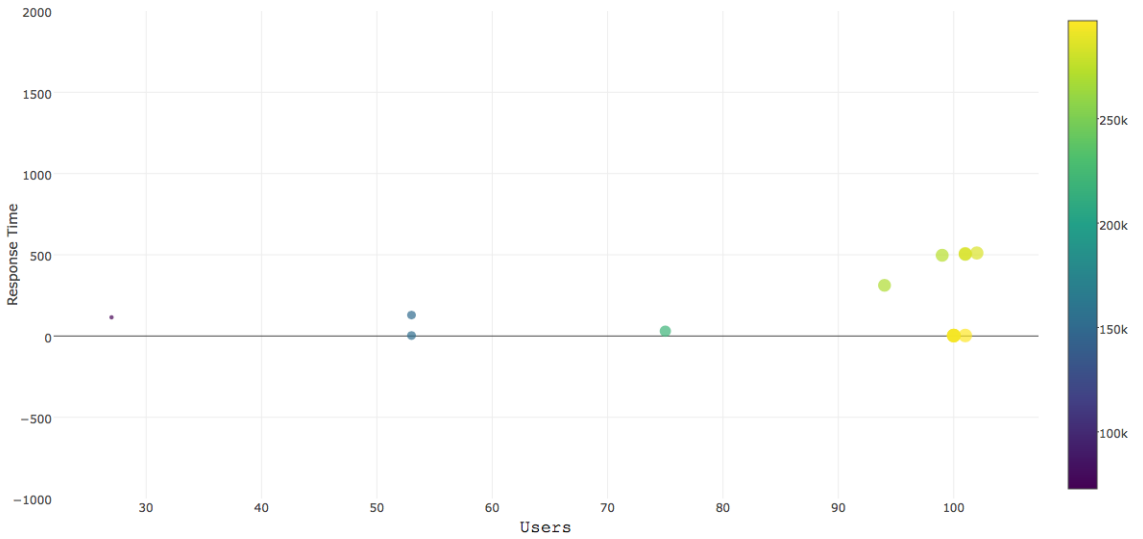


Figure 7-16: Response time by number of users of individuals with Unbalanced Processing antipattern

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 7.4 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 7.4: 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

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 suc-

ceeded in finding 29 individuals whose maximum time expected by the application was obtained. Table 7.5 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 7.5: 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			

Table 7.6 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 7.6: 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

Chapter 8

Conclusion

In this thesis we dealt with the use of hybrid metaheuristics and Q-Learning in Stress Testing.

This thesis presented a hybrid metaheuristic approach 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. Two experiments were conducted to validate the proposed approach. The first experiment was performed on an emulated component, and the second one was performed using an installed Moodle 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 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

Four experiments were performed to validate the hybrid metaheuristic and two experiments were conducted to validate the Testbed tool. The experiments uses genetic, algorithms, tabu

search, simulated annealing and the hybrid approach.

The first experiment was performed on an emulated component, and the second experiment was performed using an installed Moodle application. The collaborative approach obtained better fit values in both experiments. 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 thrid 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 hybrid 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 individual with best fitness value has 64 users on Happy Scenario 2, 81 users on Happy Scenario 1 and a response time of 12 seconds. None of the 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 72 users on Happy Scenario 2, 30 users on Happy Scenario 1, 46 user with the antipattern Tower Babel and a response time of 11 seconds. Future works include the use of new antipatterns and more experiments with the use of the antipattern Tower Babel.

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.

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.

Appendix A

Tables

Appendix B

Figures

Bibliography

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

[2] Wasif Afzal, Richard Torkar, and Robert Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6):957–976, 2009.

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

[4] Enrique Alba and Francisco Chicano. Observations in using parallel and sequential evolutionary algorithms for automatic software testing. *Computers and Operations Research*, 35(10):3161–3183, 2008.

[5] Stefano D I Alesio, Lionel C Briand, Shiva Nejati, and Arnaud Gotlieb. Combining Genetic Algorithms and Constraint Programming. *ACM Transactions on Software Engineering and Methodology*, 25(1), 2015.

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

[7] Davide Arcelli, Vittorio Cortellessa, and Catia Trubiani. Antipattern-Based Model Refactoring for Software Performance Improvement. *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures (QoSA '12)*, pages 33–42, 2012.

[8] Arthur I Baars, Kiran Lakhotia, Tanja E J Vos, and Joachim Wegener. Search-based testing, the underlying engine of Future Internet testing. *Federated Conference on Computer Science and Information Systems (FedCSIS 2011)*, pages 917–923, 2011.

[9] C Babbar, N Bajpai, and Dk Sarmah. Web Application Performance Analysis based on Component Load Testing. *International Journal of Technology*, 2011.

[10] Cornel Barna, M Litoiu, and H Ghanbari. Autonomic load-testing framework. *International conference on Autonomi*, pages 91–100, 2011.

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

[12] Christian Blum. Hybrid metaheuristics in combinatorial optimization: A tutorial. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7505 LNCS(6):1–10, 2012.

[13] Lionel C. Briand, Yvan Labiche, and Marwa Shousha. Stress testing real-time systems with genetic algorithms. *Proceedings of the 2005 conference on Genetic and evolutionary computation - GECCO '05*, page 1021, 2005.

[14] 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.

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

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

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

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

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

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

[21] Giuseppe a. Di Lucca and Anna Rita Fasolino. Testing Web-based applications: The state of the art and future trends. *Information and Software Technology*, 48:1172–1186, 2006.

[22] 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.

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

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

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

[26] Vahid Garousi. Empirical analysis of a genetic algorithm-based stress test technique. *Proceedings of the 10th annual conference on Genetic and evolutionary computation - GECCO '08*, page 1743, 2008.

[27] Vahid Garousi. A Genetic Algorithm-Based Stress Test Requirements Generator Tool and Its Empirical Evaluation. *IEEE Transactions on Software Engineering*, 36(6):778–797, November 2010.

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

[29] Jean-Yves Gendreau, Michel and Potvin. *Handbook of Metaheuristics*, volume 157. 2010.

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

[31] 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.

[32] 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.

[33] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. *2012 34th International Conference on Software Engineering (ICSE)*, pages 156–166, June 2012.

[34] Hg Gross, Bryan F Jones, and David E Eyres. Structural performance measure of evolutionary testing applied to worst-case timing of real-time systems. *Software, IEE Proceedings-*, 147(2):25–30, 2000.

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

[36] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements , open problems and challenges for search based software testing. *8th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, (Icst), 2015.

[37] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[38] B. Jones J. Wegener, K. Grimm, M. Grochtmann, H. Sthamer. Systematic testing of real-time systems. *EuroSTAR'96: Proceedings of the Fourth International Conference on Software Testing Analysis and Review*, 1996.

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

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

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

[42] Alexander Pretschner Mark Utting and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing Verification and Reliability*, 24(8):297–312, 2012.

[43] Daniel A Menascé and George Mason. TPC-W : A Benchmark for E-commerce. (June):1–6, 2002.

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

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

[46] F. Mueller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium (Cat. No.98TB100245)*, 1998.

[47] 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.

[48] Hartmut Pohlheim, Mirko Conrad, and Arne Griep. Evolutionary Safety Testing of Embedded Control Software by Automatically Generating Compact Test Data Sequences. *Analysis*, (724):804—-814, 2005.

[49] Jakob Puchinger and R Raidl. Combining Metaheuristics and Exact Algorithms in Combinatorial Optimization : A Survey and Classification. *Artificial Intelligence and Knowledge Engineering Applications a Bioinspired Approach*, 3562:41–53, 2005.

[50] P. Puschner and R. Nossal. Testing the results of static worst-case execution-time analysis. *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, 1998.

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

[52] R Raidl. A Unified View on Hybrid Metaheuristics. *Hybrid Metaheuristics (LNCS 4030)*, pages 1–12, 2006.

[53] Corey Sandler, Tom Badgett, and TM Thomas. The Art of Software Testing. page 200, September 2004.

[54] Connie U. Smith and Lloyd G. Williams. Software performance antipatterns. *Proceedings of the second international workshop on Software and performance - WOSP '00*, pages 127–136, 2000.

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

[56] C.U. Smith and L.G. Williams. Software Performance AntiPatterns; Common Performance Problems and their Solutions. *Cmg-Conference-*, 2:797–806, 2002.

[57] 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.

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

[59] El-Ghazali Talbi. *Hybrid Metaheuristics*, volume 2. 2012.

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

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

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

[63] Xingen Wang, Bo Zhou, and Wei Li. Model-based load testing of web applications. *Journal of the Chinese Institute of Engineers*, 36(1):74–86, 2013.

[64] J Wegener and M Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15(3):275–298, 1998.

[65] Joachim Wegener, Harmen Sthamer, Bryan F Jones, and David E Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, 1997.

[66] Harmen Wegener, Joachim and Pitschinetz, Roman and Sthamer. Automated Testing of Real-Time Tasks. *Proceedings of the 1st International Workshop on Automated Program Analysis, Testing and Verification (WAPATV'00)*, 2000.

[67] Alexander Wert, Jens Happe, and Lucia Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. *Proceedings - International Conference on Software Engineering*, (May):552–561, 2013.

[68] Alexander Wert, Marius Oehler, Christoph Heger, and Roozbeh Farahbod. Automatic detection of performance anti-patterns in inter-component communications. *QoSA 2014 - Proceedings of the 10th International ACM SIGSOFT Conference on Quality of Software Architectures (Part of CompArch 2014)*, pages 3–12, 2014.