Available online at www.sciencedirect.com

ScienceDirect

Information Systems 00 (2016) 1–32

Procedia Computer Science

Implementing a testbed tool for load, performance and stress search based tests

N. Gois, P. Porfírio, A. Coelho

Abstract

Metaheuristic search techniques have been extensively used to provide solutions for a more cost-effective testing process. Metaheuristics can solve complex problems in which an optimum solution must be found among a large amount of possibilities. The use of metaheuristics in testing activities is promising because of the high number of inputs that should be tested. The use of metaheuristic search techniques for the automatic generation of test has been a burgeoning interest for many researchers in recent years. Search Based Software Testing refers to the use of meta-heuristics for the optimization of a task in the context of software testing. 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. This paper addresses the problem of comparing the use of several metaheuristics in search based tests. Since a real testbed is often extremely costly, an open source testbed simulator can represent a valid alternative to real device development and testbed deployment for academic and industrial research goals. In this paper, we propose a flexible testbed tool to evaluate various diversity combining metaheuristics in search based software testing. The testbed tool emulates test scenarios in a controled environment using mock objects and implementing performance antipatterns. Two experiments were conducted to validate the proposed approach. The experiments uses genetic, algorithms, tabu search, simulated annealing and an hybrid approach.

Keywords:

1. Introduction

Performance problems such as high response times in software applications have a significant effect on the customer's satisfaction. 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. 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 [1] [2] [3] [4].

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 [5] [6].

Search-based testing is seen as a promising approach to verifying timing constraints [5]. A common objective of a load search-based test is to find scenarios that produce execution times that violate the specified timing constraints

[7]. 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.

This paper addresses the problem of comparing the use of several metaheuristics in search based tests. When comparing a new metaheuristic to existing ones, it is advantageous to test on the problem instances already tested by previous papers. Then, results will be comparable on a by-instance basis, allowing relative gap calculations between the two heuristics. In this paper, we propose a flexible testbed tool to evaluate various diversity combining metaheuristics in search based software testing. A tool named IAdapter (www.iadapter.org, github.com/naubergois/newiadapter), a JMeter plugin for performing search-based load tests, was extended [8].

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. Differently from patterns, antipatterns look at the negative features of a software system and describe commonly occurring solutions to problems that generate negative consequences.

Two experiments were conducted to validate the proposed approach. The experiments uses genetic, algorithms, tabu search, simulated annealing and an hybrid approach proposed by Gois et al. [8].

The remainder of the paper is organized as follows. Section 2 presents a brief introduction about load, performance, and stress tests. Section 3 presents concepts about the workload model. Section 4 presents details features about common performance antipatterns. Section 5 presents concepts about search based tests. Section 6 presents concepts about metaheuristic algorithms. Section 7 presents concepts about IAdapter Testbed. Section 8 shows the results of two experiments performed using the IAdapter plugin. Conclusions and further work are presented in Section 10.

2. Load, Performance and Stress Testing

Load, performance, and stress testing are typically done to locate bottlenecks in a system, to support a performance-tuning effort, and to collect other performance-related indicators to help stakeholders get informed about the quality of the application being tested [9] [10].

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 [11]. The purpose of this assessment is to demonstrate that the system reaches its performance objectives [9].

In a load testing, the system is evaluated at predefined load levels [11]. 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 [3]. 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 [5].

The stress testing verifies the system behavior against heavy workloads [9], 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 [11] [3].

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

2.1. Stress Test Process

Contrary to functional testing, which has clear testing objectives, Stress testing objectives are not clear in the early development stages and are often defined later on a case-by-case basis. The Fig. 1 shows a common Load, Performance and Stress test process [2].

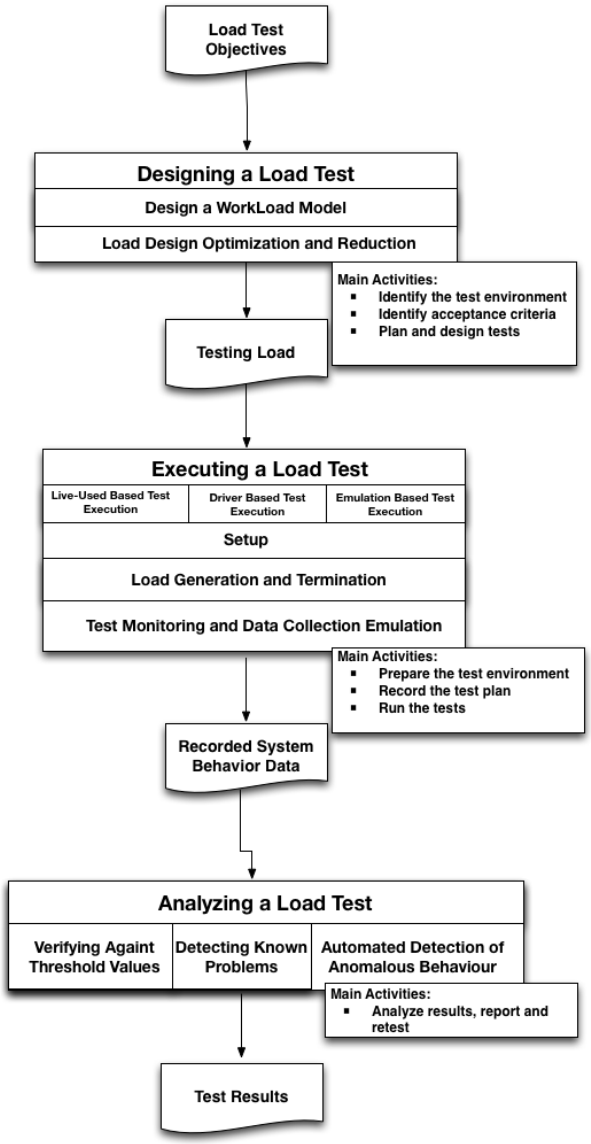

Figure 1. Load, Performance and Stress Test Process [2][12]

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 [2].

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[2].

The core activities in conducting an usual Load, Performance and Stress tests are [12]:

- Identify the test environment: identify test and production environments and knowing the hardware, software, and network configurations helps derive an effective test plan and identify testing challenges from the outset.

- Identify acceptance criteria: identify the response time, throughput, and resource utilization goals and constraints.
- Plan and design tests:identify the test scenarios.In the context of testing, a scenario is a sequence of steps in an application. It can represent a use case or a business function such as searching a product catalog, adding an item to a shopping cart, or placing an order [10].
- Prepare the test environment: configure the test environment, tools, and resources necessary to conduct the planned test scenarios.
- Record the test plan: record the planned test scenarios using a testing tool.
- Run the tests: Once recorded, execute the test plans under light load and verify the correctness of the test scripts and output results.
- Analyze results, report, and retest: examine the results of each successive run and identify areas of bottleneck that need addressing.

2.2. Automated Stress Test Tools

Automated tools are needed to carry out serious load, stress, and performance testing. Sometimes, there is simply no practical way to provide reliable, repeatable performance tests without using some form of automation. The aim of any automated test tool is to simplify the testing process. Automated Test Tool typically have the following components [3]:

- Scripting module: Enable recording of end-user activities in different middleware protocols;
- Test management module: Allows the creation of test scenarios;
- Load injectors: Generate the load with multiple workstations or servers;
- Analysis module: Provides the ability to analyse the data collected by each test interation.

Apache JMeter is a free open source stress testing tool. It has a large user base and offers lots of plugins to aid testing. JMeter is a desktop application designed to test and measure the performance and functional behavior of applications. The application it's purely Java-based and is highly extensible through a provided API (Application Programming Interface). JMeter works by acting as the client of a client/server application. JMeter allows multiple concurrent users to be simulated on the application [13] [12].

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

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

2.3. Stress Test Results

The system behavior recorded during the test execution phase needs to be analyzed to determine if there are any load-related functional or non-functional problems [2].

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 [2].

3. WorkLoad Model

Load, performance, or stress testing projects should start with the development of a model for user workload that an application receives. This should take into consideration various performance aspects of the application and the infrastructure that a given workload will impact. A workload is a key component of such a model [3].

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 [14] [3] [15].

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 [14] [3].

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 [14] [3].

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 [11].

In descriptive models, one finds different levels of abstraction on the one hand and different levels of fidelity to the original data on the other hand. The most strictly faithful models try to mimic the data directly using the statistical distribution of the data. The most common strategy used in descriptive modeling is to create a statistical model of an observed workload (Fig. 2). This model is applied to all the workload attributes, e.g., computation, memory usage, I/O behavior, communication, etc. [11]. Fig. 2 shows a simplified workflow of a descriptive model. The workflow has six phases. In the first phase, the user uses the system in the production environment. In the second phase, the tester collects the user's data, such as logs, clicks, and preferences, from the system. The third phase consists in developing a model designed to emulate the user's behavior. The fourth phase is made up of the execution of the test, emulation of the user's behavior, and log gathering.

Generative models are indirect in the sense that they do not model the statistical distributions. Instead, they describe how users will behave when they generate the workload. An important benefit of the generative approach is that it facilitates manipulations of the workload. It is often desirable to be able to change the workload conditions as part of the evaluation. Descriptive models do not offer any option regarding how to do so. With the generative models, however, we can modify the workload-generation process to fit the desired conditions [11]. The difference between the workflows of the descriptive and the generative models is that user behavior is not collected from logs, but simulated from a model that can receive feedback from the test execution (Fig. 3).

Both load model have their advantages and disadvantages. In general, loads resulting from realistic-load based design techniques (Descriptive models) can be used to detect both functional and non-functional problems. However, the test durations are usually longer and the test analysis is more difficult. Loads resulting from fault-inducing load design techniques (Generative models) take less time to uncover potential functional and non-functional problems, the resulting loads usually only cover a small portion of the testing objectives [2]. The presented research work uses a generative model.

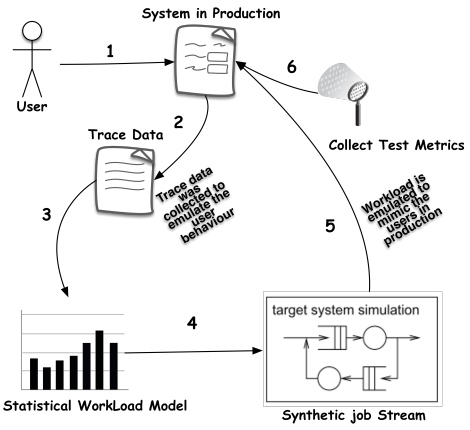

Figure 2. Workload modeling based on statistical data [11]

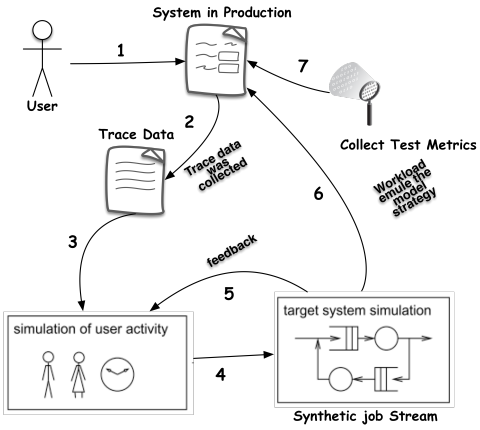

Figure 3. Workload modeling based on the generative model [11]

4. Common performance application problems and performance antipatterns

Performance is critical to the success of today's software systems. Many software products fail to meet their performance objectives when they are initially constructed. Performance problems share common symptoms and many performance problems described in the literature are defined by a particular set of root causes. Fig. 4 shows the symptoms of known performance problems [16].

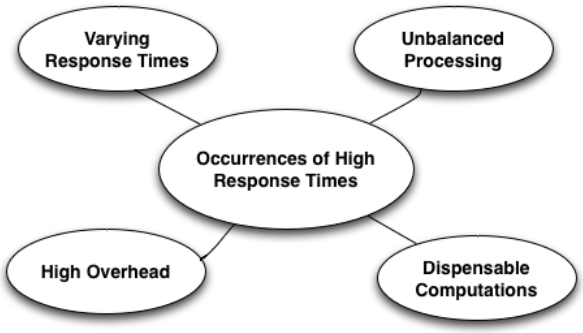

Figure 4. Symptoms of know performance problems [16].

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 [17]. The table 1 present some of the most common performance antipatterns.

Table 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 [18] [19].

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 [19]. The Figures 5 and 6 describes an hypotetical system with a BLOB problem: The Fig. 5 presents a sample where the Blob class uses the features A,B,C,D,E,F and G of the hypotetical system; The Fig. 6 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 [20][16].

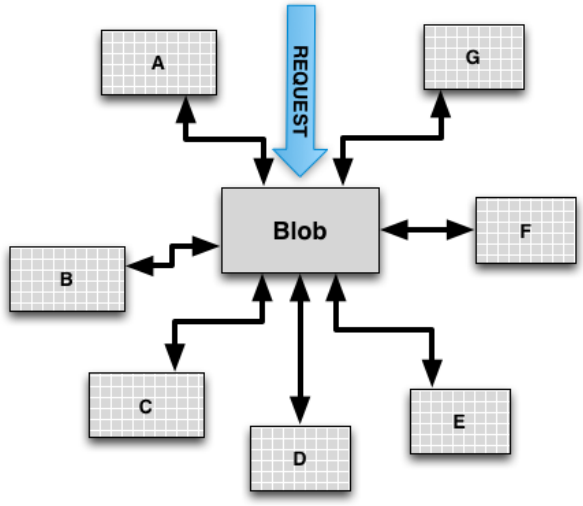

Figure 5. The God class[16].

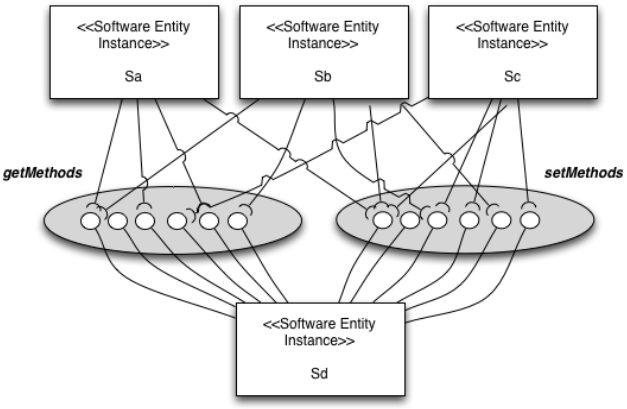

Figure 6. The God class[20].

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

processing it is when processes cannot make effective use of available processors either because processors are dedicated to other tasks or because of single-threaded code. This manifestation has available processors and we need to ensure that the software is able to use them. Fig. 7 shows a sample of the Unbalanced Processing. In The Fig. 7, 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. 8 describes a software S with a Pipe and Filter Architectures problem: (a) Static View, there 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. 9 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 [20].

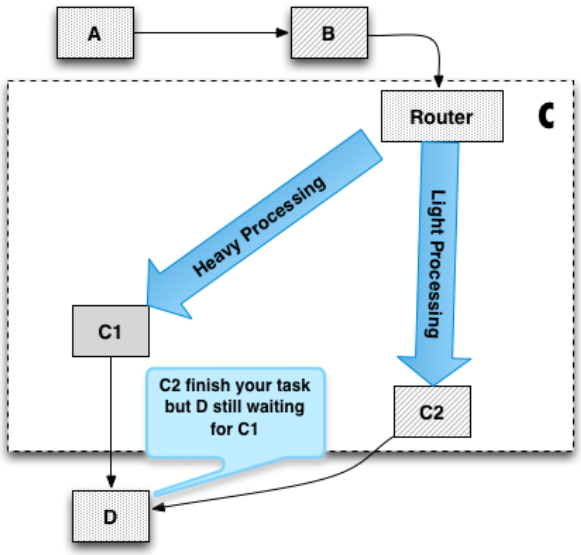

Figure 7. Unbalanced Processing sample [16].

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 [21] [22]. Circuitous Treasure Hunt are typical performance antipatterns that causes unnecessarily frequent database requests. The Circuitous Treasure Hunt antipattern is a result from a bad database schema or query design. A common Circuitous Treasure Hunt design creates a data dependency between single queries. For instance, a query requires the result of a previous query as input. The longer the chain of dependencies between individual queries the more the Circuitous Treasure Hunt antipattern hurts performance [4]. The Fig. 10 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 [20].

Empty Semi Trucks occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both [23]. There are a special case of Empty Semi

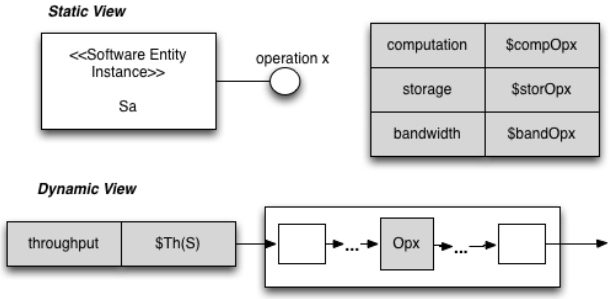

Figure 8. Pipe and Filter sample [20]

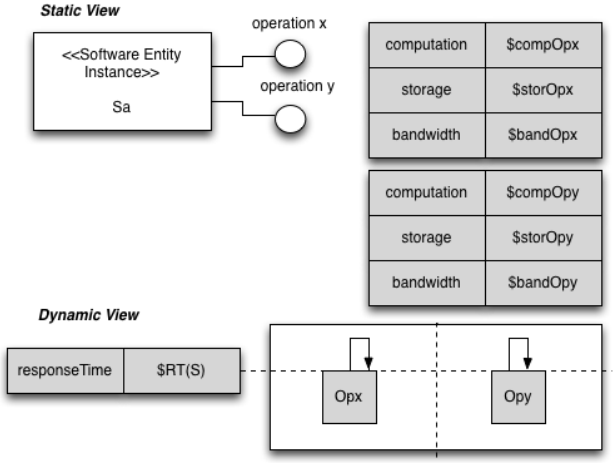

Figure 9. Extensive Processing sample [20].

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 [20].

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 [22]. 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 Sd performs many times the translation of format for communicating with other instances [20].

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 capacity 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 [20].

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 [21] [22].

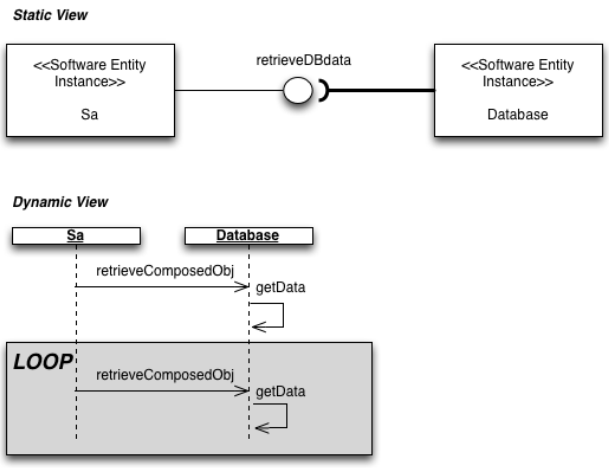

Figure 10. Circuitous Treasure Hunt sample [20]

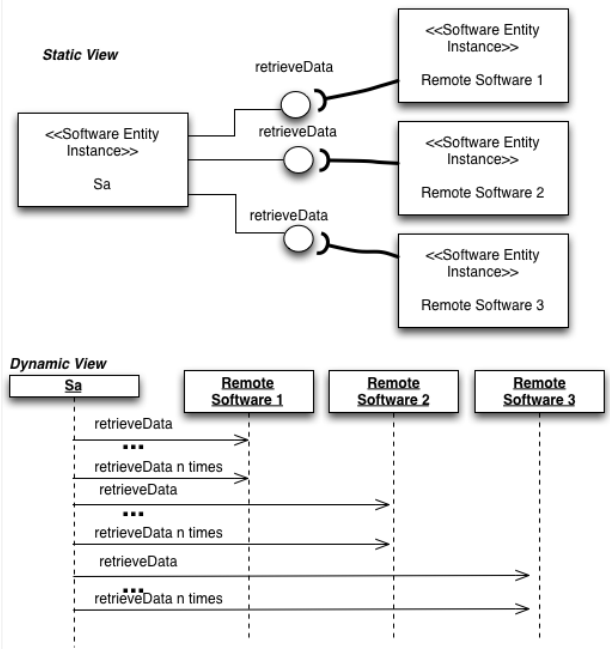

Figure 11. Empty Semi Trucks sample [20].

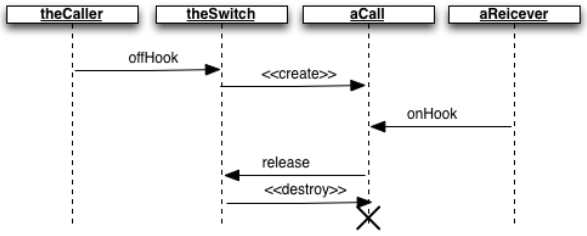

Figure 14. Excessive Dynamic Allocation.

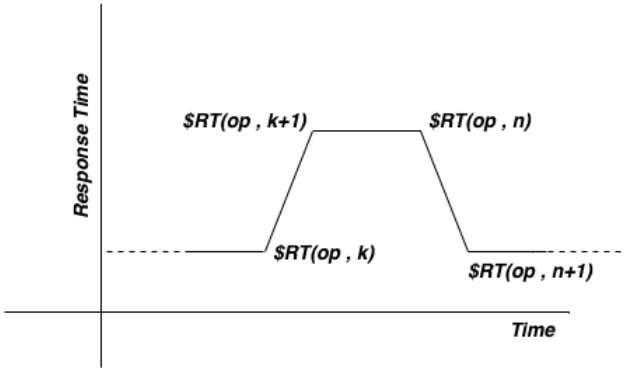

Figure 15. Traffic Jam Response Time [20].

The Fig. 14 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 15 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 [20].

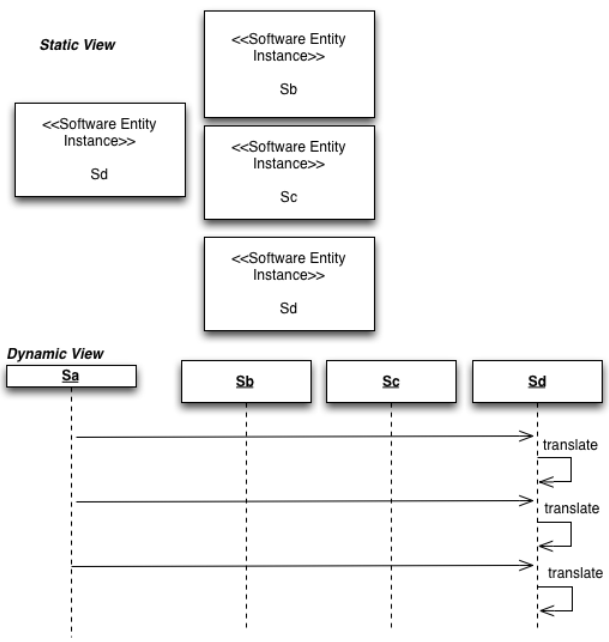

Figure 12. Tower of Babel sample [20]

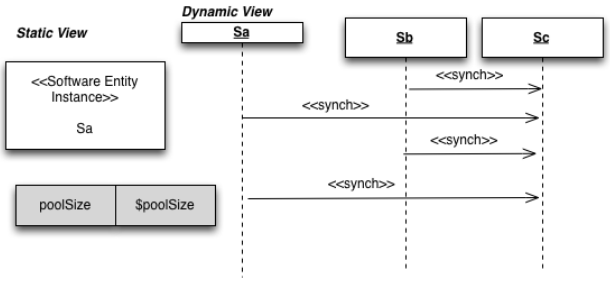

Figure 13. One-Lane Bridge sample [20].

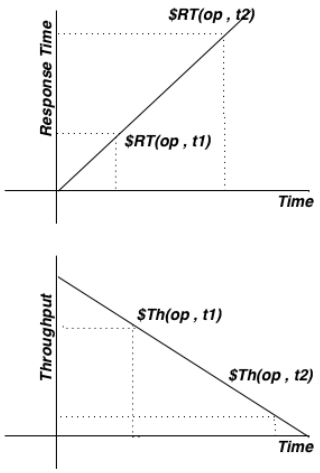

Figure 16. The Ramp sample [20].

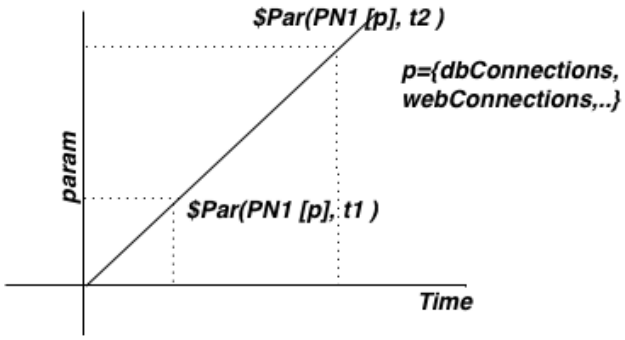

Figure 17. More is Less sample [20].

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 [4] [22]. Fig. 16 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. 17 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.

To emulate the presented antipatterns the testbed solution uses Mock Objects with the JMeter load test tool.

5. Mock Objects

Many times, a unit test should test a class in isolation. Sometimes, side effects from other classes or the system should be eliminated. To eliminate these side effects it is necessary to replace dependencies to other classes.

A mock object is a dummy implementation for an interface or a class. A Mock Object is a substitute implementation to emulate other domain code. Basic mock object allows testing a unit faking the communication with collaborating objects. It should be simpler than the real code, not duplicate its implementation [24] [25]. There are several reasons to use Mock Objects [26]:

- The real object has nondeterministic behavior.
- The real object is difficult to set up.
- The real object has behavior that is hard to trigger.
- The real object is slow.
- The real object has a user interface.
- The test needs to ask the real object about how it was used.
- The real object does not yet exist.

A mock object can be created manually or using a mock framework. Mock frameworks allows the creation of a mock object at runtime and define their behavior. Mockito is a mock framework which can be used in conjunction with JUnit. Mockito allows to create and configure mock objects.

```java
import static org.mockito.Mockito.*;

public class MockitoTest  {

        @Mock
        MyDatabase databaseMock; (1)

        @Rule public MockitoRule mockitoRule = MockitoJUnit.rule();
(2)

        @Test
        public void testQuery()  {
                ClassToTest t  = new ClassToTest(databaseMock); (3)
                boolean check = t.query("* from t"); (4)
                assertTrue(check); (5)
                verify(databaseMock).query("* from t"); (6)
        }
}
```

Figure 18. A mockito test sample extracted from http://www.vogella.com/tutorials/Mockito/article.html

Fig. 18 shows a sample fo unit test using the framework Mockito extracted from http://www.vogella.com/tutorials/Mockito. The item 1 tells Mockito to mock the databaseMock instance.

6. Search Based Tests

Search-based software engineering (SBSE) is the application of optimization techniques in solving software engineering problems [1,2]. The applicability of optimization techniques in solving software engineering problems is suitable as these problems frequently encounter competing constraints and require near optimal solutions [5] [27].

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 [5] [28] [27].

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 [5]:

- 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. 19 shows a comparison between the range of metaheuristcs and the type of non-functional search based test.The Data comes from Afzal et al. [29]. Afzal's work adds to some of the latest research in this area ([30] [31] [32] [33] [34] [8]).

6.1. Load, Stress and Performance Search Based Testing

A common goal of load, performance and stress search-based testing 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 [7]. 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 [7]. 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 [5].

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 [5] [35].

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 [5]. However, stress testing where testers have no access to the production environment should be measured by the execution time measurement [3] [5].

Table 2 shows a comparison between the research studies on load, performance, and stress tests presented by Afzal et al. [29]. Afzal's work adds to some of the latest research in this area ([30] [31] [32] [33] [34] [8]). The columns represent the type of tool used (prototype or functional tool), and the rows represent the metaheuristic approach used

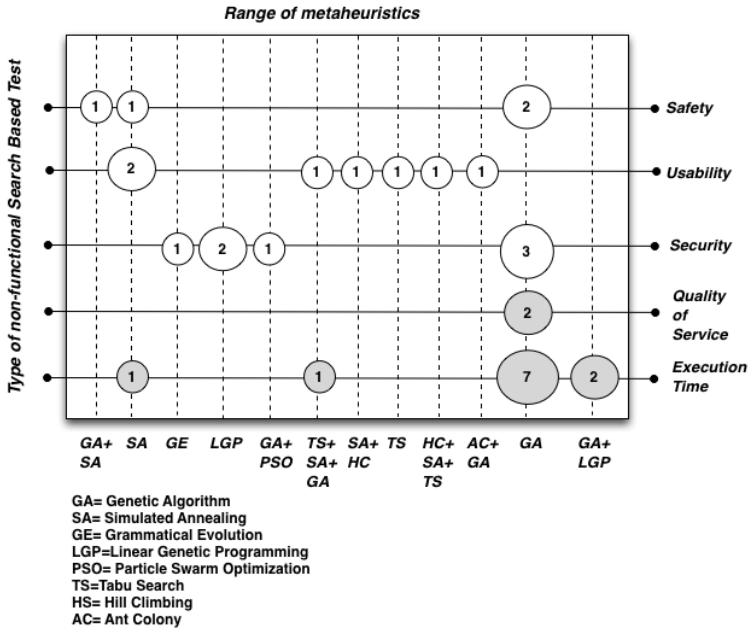

Figure 19. Range of metaheuristics by Type of non-functional Search Based Test[5].

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. [37] 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 [37].

Alander et al. [36] 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 generated more input cases with longer response times [36].

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] [41] .

Gro et. al. [45] 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 [45].

Tracey et al. [48] 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 [48].

Pohlheim and Wegener used an extension of genetic algorithms with multiple sub-populations, each using a different search strategy. The duration of execution measured in processor cycles was taken as the fitness function. The GA found longer execution times for all the given modules in comparison with systematic testing[49].

Briand et al. [39] 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 [39].

Di Penta et al. [46] 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

Table 2. 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 [8]
GA	Alander et al.,1998 [36] Wegener et al., 1996 and 1997 [37][38] Sullivan et al., 1998 [7] Briand et al., 2005 [39] Canfora et al., 2005 [40]	Wegener and Grochtmann, 1998 [41] Mueller et al., 1998 [42] Puschner et al. [43] Wegener et al., 2000 [44] Gro et al., 2000 [45]	Di Penta, 2007 [46] Garoussi, 2006 [30] Garousi, 2008 [47] Garousi, 2010 [31]
Simulated Annealing (SA)			Tracey, 1998 [48]
Constraint Programming			Alesio, 2014 [33] Alesio, 2013 [32]
GA + Constraint Programming			Alesio, 2015 [34]
Customized Algorithm		Pohlheim, 1999 [49]	

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 [46].

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 [30].

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 [34].

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 [8].

7. 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 [50].

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 [51]:

- 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 [52].

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 [50].

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 [53].

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 [54].

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 [50].

Algorithm 3 Genetic Algorithm

$s \leftarrow GenerateInitialSolution()$
Evaluate(P)
while termination conditions not met **do**
  $P_1 \leftarrow Recombine(P)$
  $P_2 \leftarrow Mutate(P_1)$
  $Evaluate(P_2)$
  $P \leftarrow Select(P_2, P)$
end while

7.1. Hybrid Metaheuristc

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 [50].

There are two main categories of metaheuristic combinations: collaborative combinations and integrative combinations. 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 successful [50].

8. Testbed systems

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 [55].

TPC-W specifies an Ecommerce workload that simulates the activities of a retail store website. Emulated users can browse and order products from the website. A user is emulated via a Remote Browser Emulator, RBE, that simulates the same HTTP network traffic as would be seen by a real customer using a browser. In fact one can connect a real browser and walk through the TPC-W website browsing and ordering books.

TPC-W presents some metrics to evaluate test scenarios:

- Response Time (WIRT): WIRT is defined by TPC-W as t2-t1, where t1 is the time measured at the Emulated Browsers and t2 is the time when the last byte of the last HTTP response that completes the web interaction is received.

Bertolino et al. present a model-based approach to generate stubs for Web Services which respect both an extra-functional contract expressed via a Service Level Agreement [56].

Raúl et al. presents a testbed approach that uses the benchmark TPC-W. The approach uses a workload generator named GUERNICA. GUERNICA is an universal Generator of Dynamic Workload under WWW Platforms, which is a web workload generator and testing tool to evaluate performance and functionality of web applications.

9. The IAdapter Testbed system

In this section, We devise a new testbed that has the ability to reproduce different types of web workloads. The proposed solution extends a tool named IAdapter to create a testbed tool to validade load, performance and stress search based tests approaches.

The IAdapter is a JMeter plugin designed to perform search-based stress tests. Fig. 20 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 [8].

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.

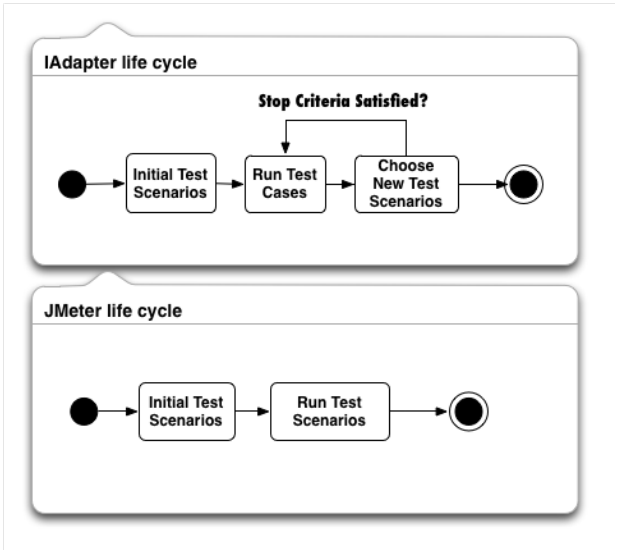

Figure 20. IAdapter life cycle

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 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. The Fig. 21 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.

includegraphics[width=0.6]./images/testbedarch.png

Figure 21. testbed main architecture.

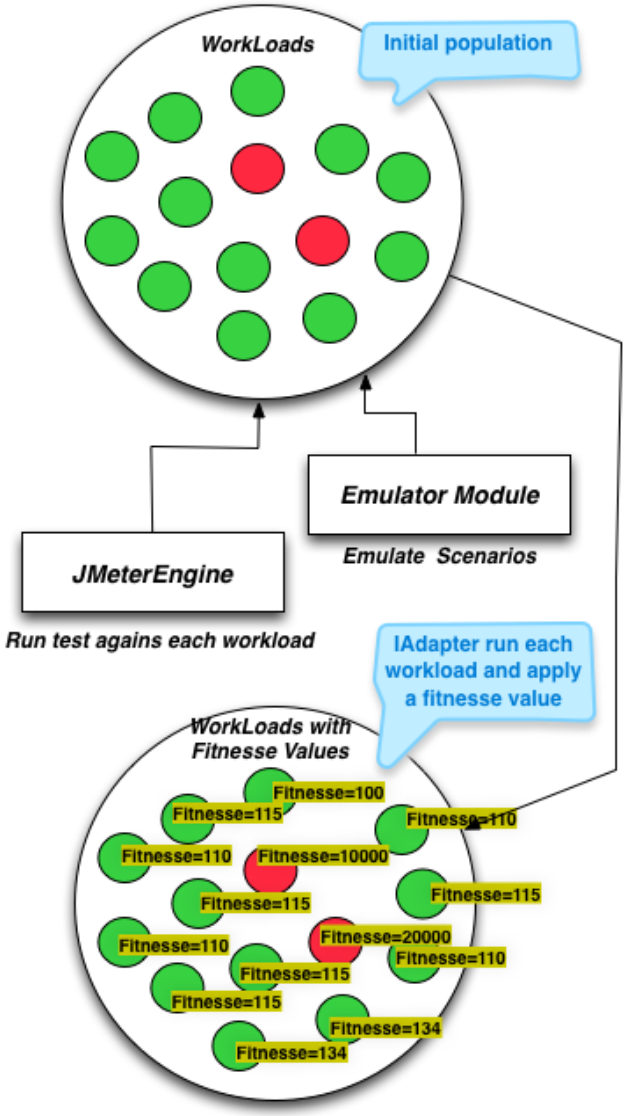

Figure 22. Test Module first feature.

9.1. 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 successful scenarios and antipatterns implementations. The heuristics are executed in order to select the scenarios with failures or high response times. The Fig. 22 presents the first feature of Test Module where a initial population it is created and IAdapter with JMeterEngine performs all tests and apply a fitnesse value to each workload.

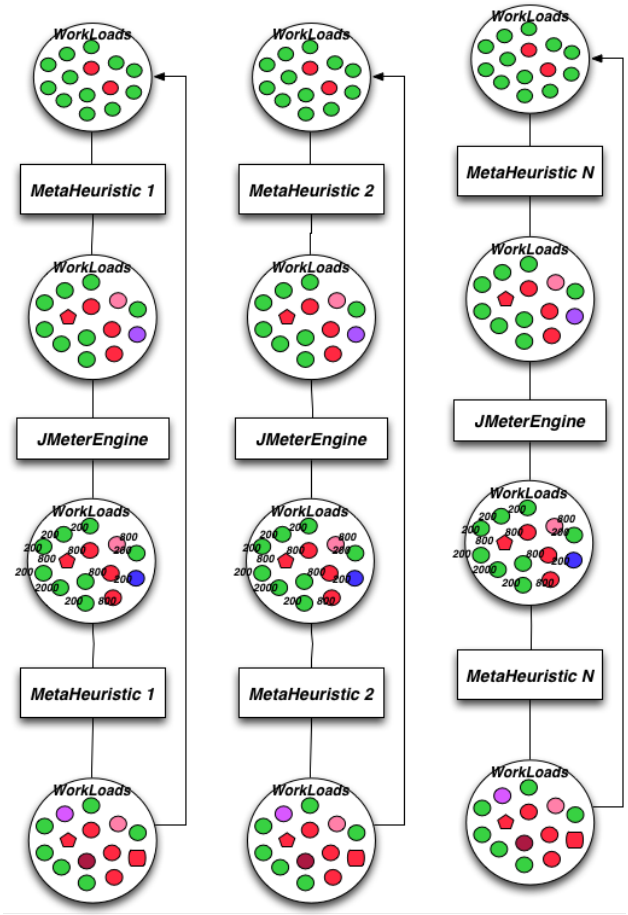

Figure 23. Test Module life cycle.

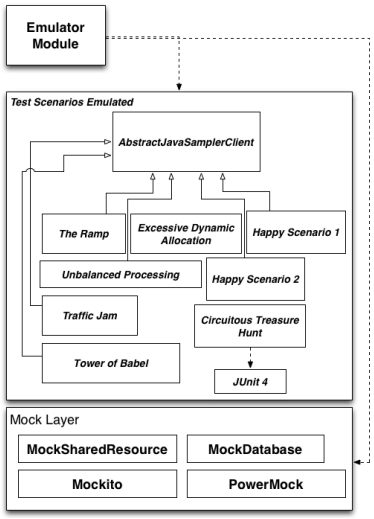

Figure 24. Heuristic class diagram.

The Fig. 23 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 fitnesse 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:

$$\begin{aligned} fitnesse = {} & 90percentileweigth * 90percentiletime \\ & +80percentileweigth * 80percentiletime \\ & +70percentileweigth * 70percentiletime+ \\ & maxResponseWeigth * maxResponseTime+ \\ & numberOfUsersWeigth * numberOfUsers - penalty \end{aligned} \quad (1)$$

The use of presented fitnesse 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. 25 shows the class diagram for custom and provided heuristics. All heuristic classes extends 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.

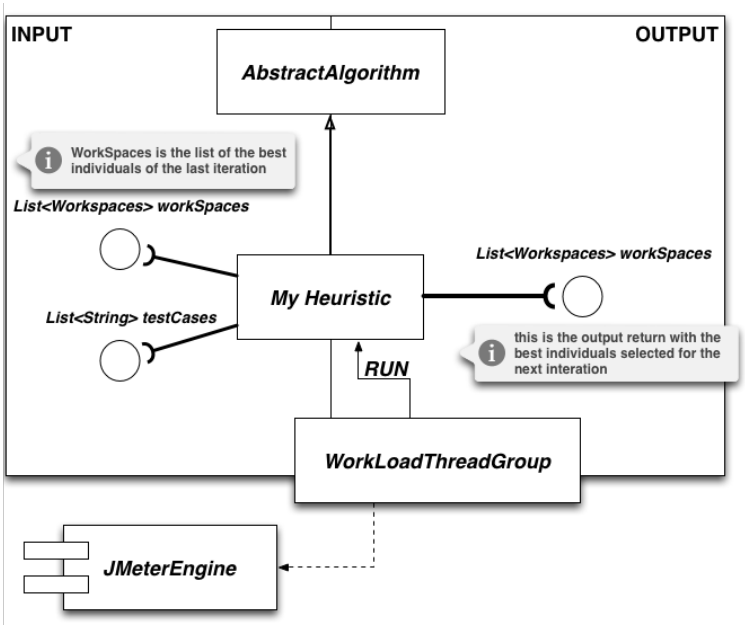

Figure 25. Heuristic class diagram.

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

9.2. 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. 26 presents the main features of the emulator module. The module implements 8 test scenarios in its first version.

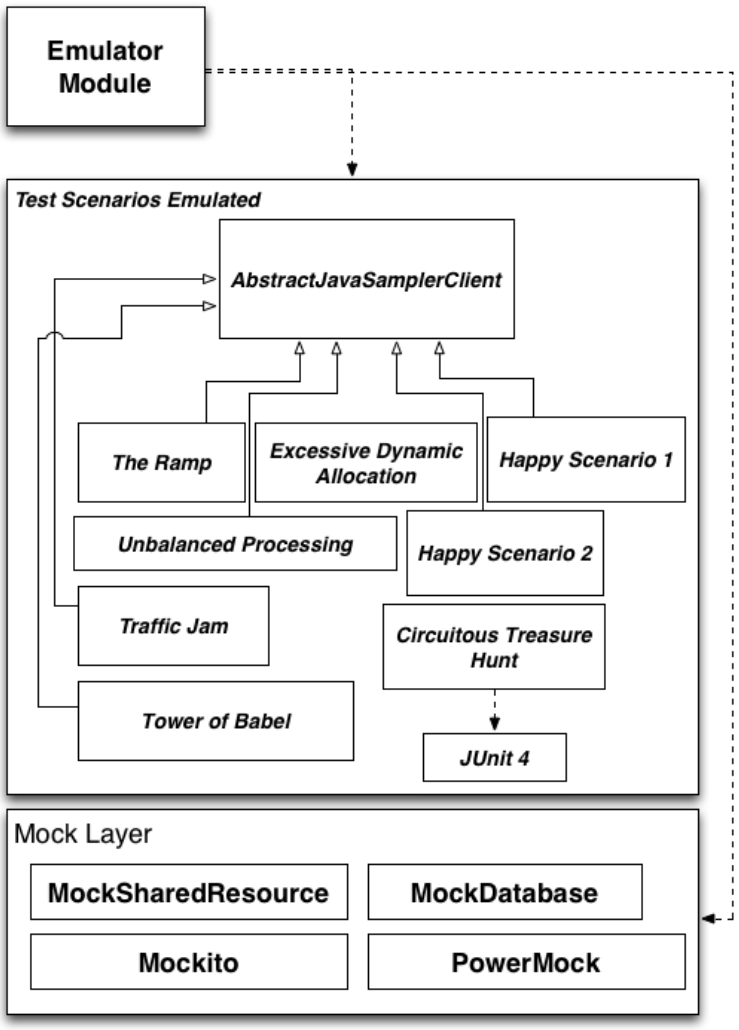

Figure 26. Heuristic class diagram.

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: if users > 10 and users<20 then
2:     sleep(10*users)
3: end if
4: if users >= 20 and users<30 then
5:     sleep(20*users)
6: end if
7: if users >= 30 and users<40 then
8:     sleep(30*users)
9: end if
10: if users >= 40 and users<50 then
11:     sleep(40*users)
12: end if
13: if users >= 50 and users<60 then
14:     sleep(60*users)
15: end if
16: if users >= 60 and users<70 then
17:     sleep(70*users)
18: end if
19: if users >= 80 and users<90 then
20:     sleep(80*users)
21: end if
22: if users >= 90  then
23:     sleep(90*users)
24: end if
```

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

9.3. Test Scenario Representation Module

This modules provides a set of scenarios in a common representation. The representation of a scenario 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. 27 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 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.

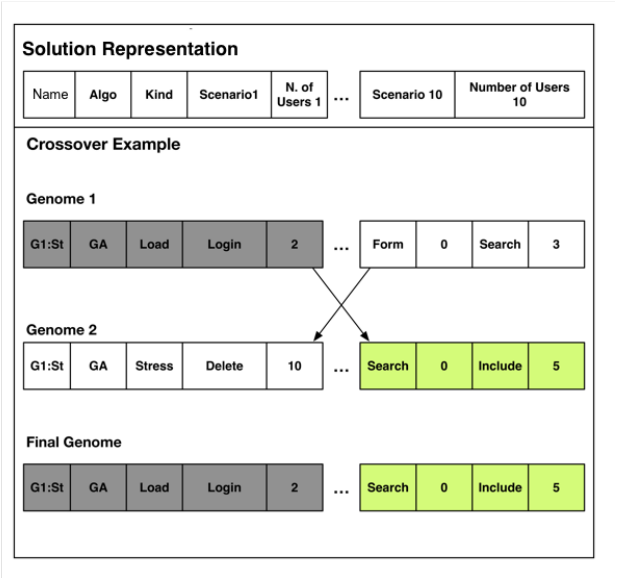

Figure 27. Solution representation and crossover example

9.4. Neighborhood provider service

Fig. 28 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.

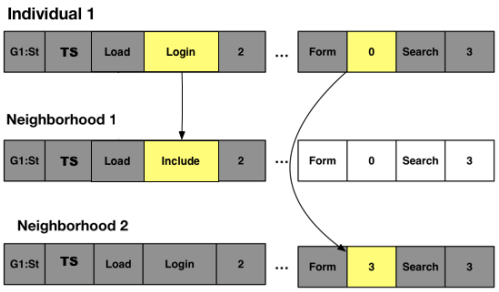

Figure 28. Neighborhood provider strategy

10. Experiments

In this section, We present the results of experiments which we carried out to verify the antipatterns implementation, the fitnesse objective function and the metaheuristics used by the testbed tool. 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 individuals 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 used the following fitness (goal) function:

$$\begin{aligned} fit = {} & 3000 * numberOfUsers \\ & -20 * 90percentiletime \\ & -20 * 80percentiletime \\ & -20 * 70percentiletime \\ & -20 * maxResponseTime \\ & -penalty \end{aligned} \tag{2}$$

The objective function applied is intended to maximize the number of users and minimize the response time of the scenarios being tested. A penalty is applied when the response time is greater than the maximum response time expected. The penalty is calculated by the follow equation:

$$\begin{aligned} penalty &= 100 * \Delta \\ \Delta &= (t_{CurrentResponseTime} - t_{MaximumResponseTimeExpected}) \end{aligned} \tag{3}$$

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.

to find the maximum number of users with mito verify if the emulated antipatterns could be applied in a testbed. The next subsections present details about the two experiments.

10.1. The Ramp and Circuitous Treasure Hunt experiment

This experiment was performed in six 20 generations with three scenarios (The Ramp, Circuitous Treasure Hunt and Happy Scenario 1). The experiment has the following goals:

- (Goal 1) find the scenarios whose response time has a value of up to 5 seconds.

- (Goal 2) Identify and penalize the scenarios that contains the Ramp or Circuitous Treasure Hunt antipattern.
- (Goal 3) Identify the algorithms with better fitnesse value and response time.

The experiment uses tabu search, hill climbing, genetic algorithms and the hybrid metaheuristic approach proposed by Gois et al. [8]. Scenarios were generated with the Ramp and Circuitous Treasure antipattern as well as scenarios with Happy Scenario 1 and mixed scenarios. The Fig. 29 presents the fitnesse value obtained by The Hybrid metaheuristic approach in comparasion with Hill Climbing metaheuristic. The Fig. 30 presents the comparasion between Hybrid metaheuristic and genetic algorithms. The Fig. 31 presents the comparasion between Hybrid metaheuristic and Tabu Search. The Hybrid scenario ussually obtained better fitnesse value than all the others metaheuristics (Goal 3). The higher the fit value, the average response time it is closer to 5 seconds (Fig. 32).

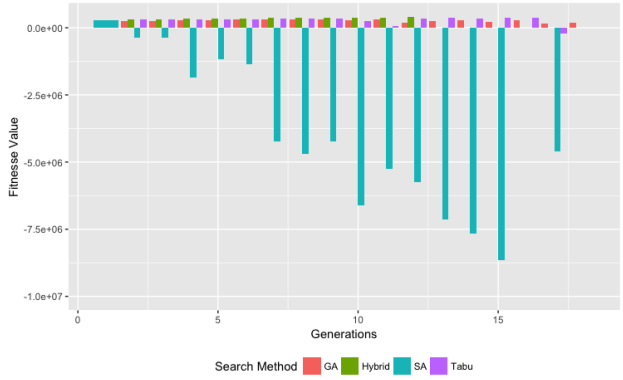

Figure 29. Comparison between Hybrid and Hill Climbing algorithms

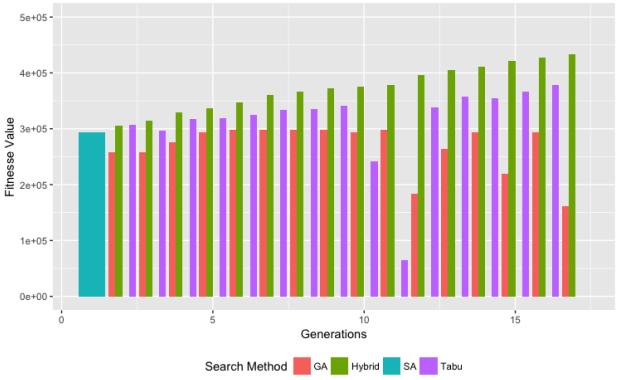

Figure 30. Comparison between Hybrid and Genetic algorithms

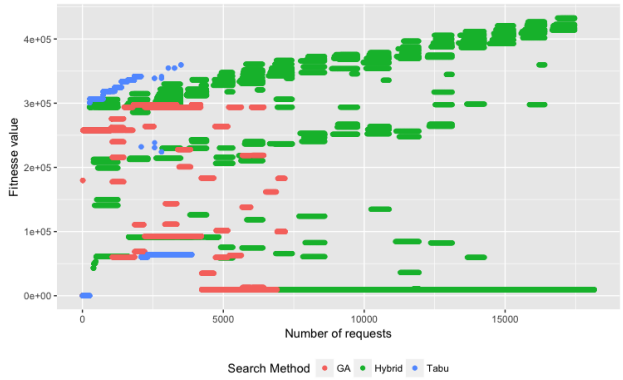

Figure 31. Comparison between Hybrid and Tabu Search algorithms

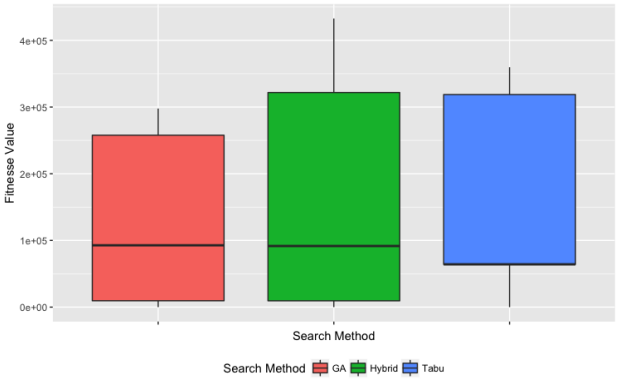

Figure 32. Fitnesse value by response time

The Fig. 33 presents the average response time value by generation of the tests that have only the Happy Scenario 1. The Fig. 34 presents the fitnesse value by generation of the tests that have only the Happy Scenario 1. The response time and fitnesse value is linearly increased over the course of the new generations until tests finds workloads with 5 seconds of response time.

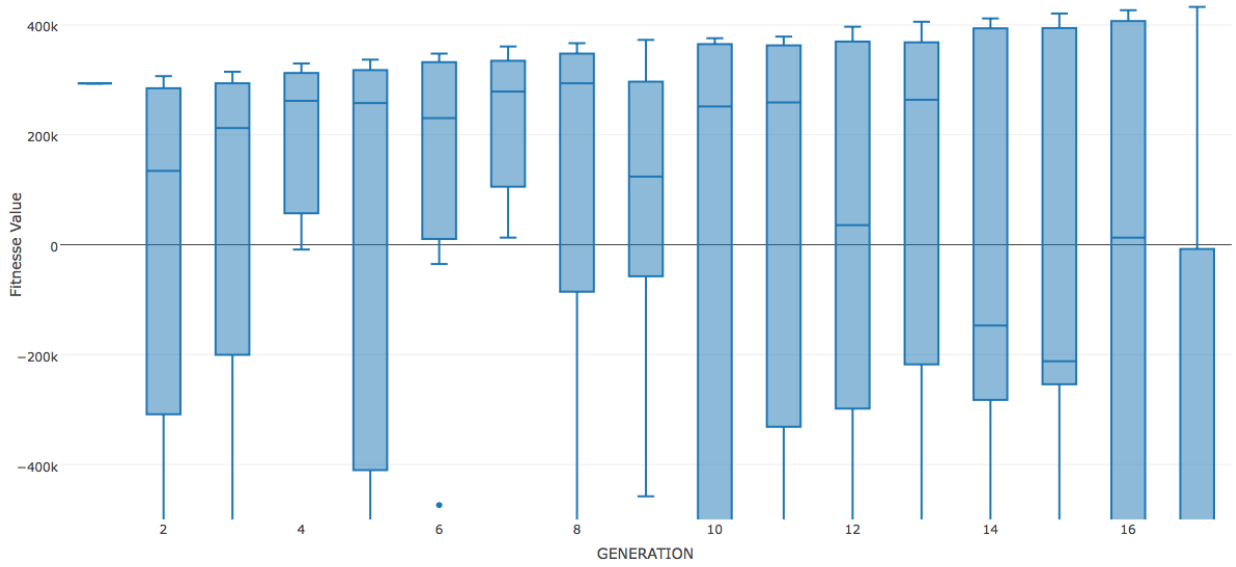

Figure 33. Response time by generation in tests without the Ramp scenario

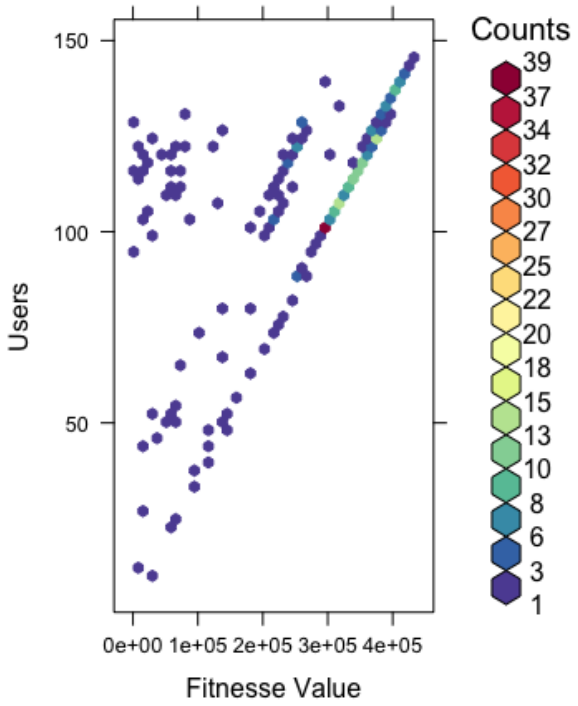

Figure 34. Fitnesse value by generation in tests without the Ramp scenario

Table **??** shows 9 tests where the response time of 5 seconds is reached. While the happy scenario reaches the

Table 3. My caption

Search Method	Generation	Users	Fitnesse 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

time of 5 seconds with a range of 71 to 75 users (Test 4 to 8), the implementation of the antipattern The Ramp requires only 2 users with 7 to 11 users of the happy scenario.

Fig. 40 presents the response time by generation with and without antipatterns scenarios. The Fig. 41 presents the response time by generation with and without antipatterns scenarios. The red squares are tests with antipattern scenarios. The green squares are tests without the antipattern scenarios. It is possible to observe that the tests with no anti pattern scenarios has response time up to 6 seconds and the scenarios with The Ramp and Circuitous Treasure Hunt antipatterns was penalized with negative values to fitnesse value (Goal 2).

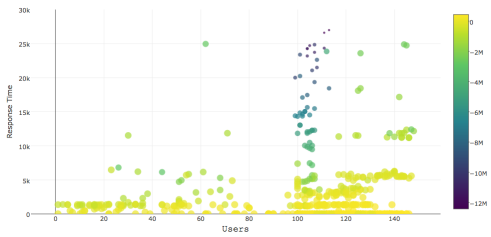

Figure 35. Response time by generation in all tests scenarios

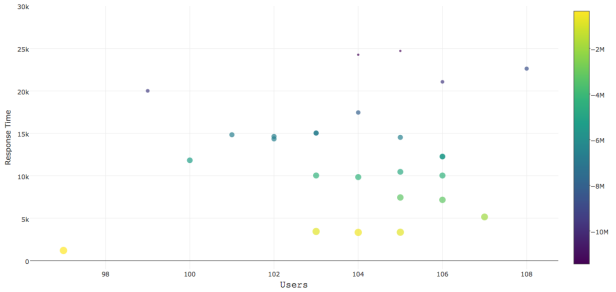

Figure 36. Finesse value by generation in all tests

10.2. The Tower Babel and Unbalanced Processing experiment

This experiment was performed in six 20 generations with four scenarios (Tower Babel, Unbalanced Processing, Happy Scenario 1 and Happy Scenario 2). Fig. 38 presents the fitnesse value obtained by number of requests of each metaheuristic. The Fig. 43 presents the response time by user with antipatterns scenarios. The test scenarios in Fig. 42

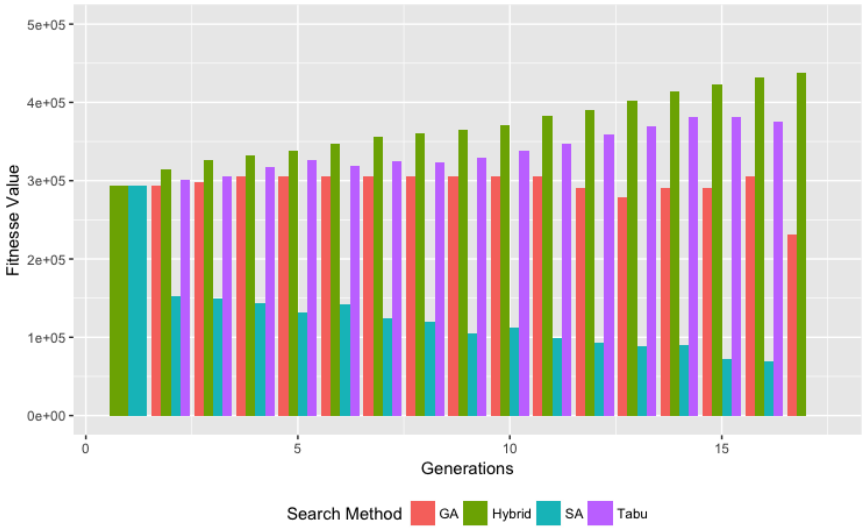

Figure 37. Response time by generation in all tests scenarios

The red squares are tests with antipattern scenarios. The green squares are tests without the antipattern scenarios. It is possible to observe that the tests with no anti pattern scenarios has response time up to 6 seconds and the scenarios with The Ramp and Circuitous Treasure Hunt antipatterns was penalized with negative values to fitnesse value (Goal 2).

Figure 38. Response time by generation in all tests scenarios

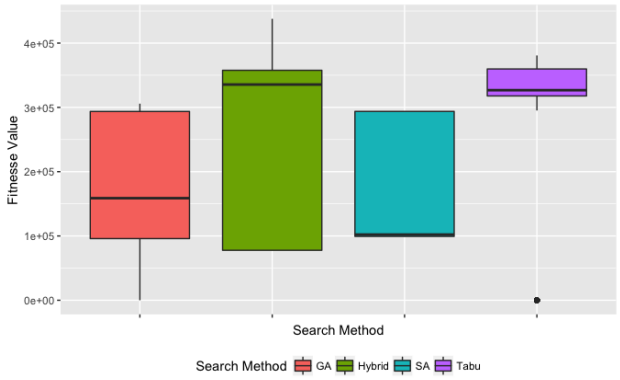

Figure 39. Finesse value by generation in all tests

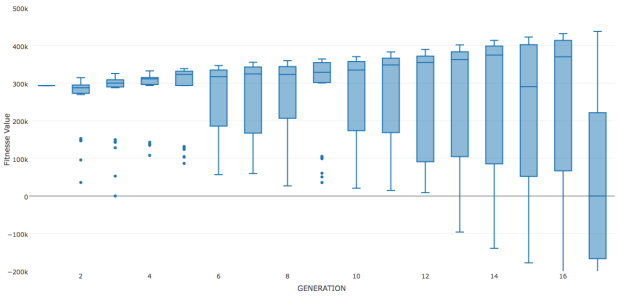

Figure 40. Response time by generation in all tests scenarios

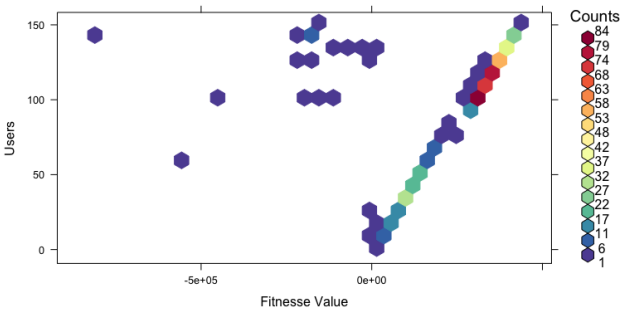

Figure 41. Finesse value by generation in all tests

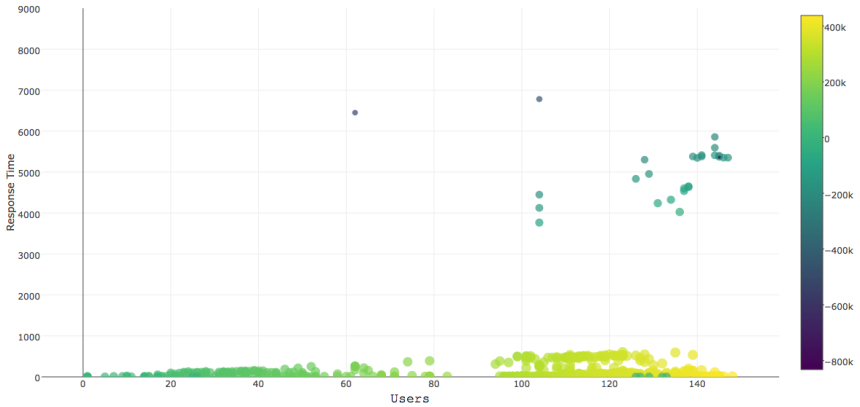

Figure 42. Response time by generation in all tests scenarios

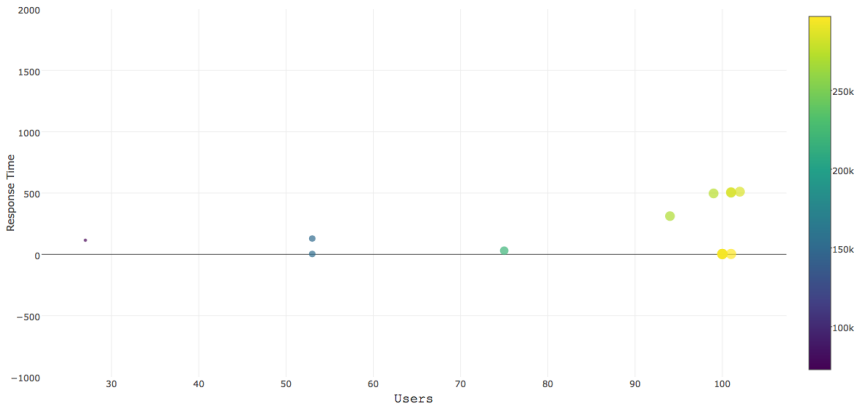

Figure 43. Finesse value by generation in all tests

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 $p \leq 0.05$ was 89.The result was significant at $p \leq 0.05$.

Table 4. My caption

Search Method	Generation	Users	Fitnesse 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

The procedure showed that there was a significant improvement in the results with the use of the Hybrid Metaheuristc .

11. Conclusion

References

[1] D. Draheim, J. Grundy, J. Hosking, C. Lutteroth, G. Weber, Realistic load testing of Web applications, in: Conference on Software Maintenance and Reengineering (CSMR'06), 2006. doi:10.1109/CSMR.2006.43.

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

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

[4] A. Wert, M. Oehler, C. Heger, R. 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) (2014) 3–12doi:10.1145/2602576.2602579.
URL http://dx.doi.org/10.1145/2602576.2602579

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

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

[7] M. O. Sullivan, S. Vössner, J. Wegener, D.-b. Ag, Testing Temporal Correctness of Real-Time Systems — A New Approach Using Genetic Algorithms and Cluster Analysis — 1–20.

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

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

[10] M. Corporation, Performance Testing Guidance for Web Applications (Nov. 2007).
URL http://www.amazon.com/Performance-Testing-Guidance-Web-Applications/dp/0735625700http://msdn.microsoft.com/en-us/library/bb924375.aspx

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

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

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

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

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

[16] A. Wert, J. Happe, L. Happe, Supporting swift reaction: Automatically uncovering performance problems by systematic experiments, Proceedings - International Conference on Software Engineering (May) (2013) 552–561. doi:10.1109/ICSE.2013.6606601.

[17] W. H. Brown, R. C. Malveau, H. W. McCormick, T. J. Mowbray, AntiPatterns: refactoring software, architectures, and projects in crisis, John Wiley & Sons, Inc., 1998.

[18] V. Cortellessa, L. Frittella, A Framework for Automated Generation of Architectural Feedback from Software Performance Analysis (2007) 171–185.

[19] C. U. Smith, L. G. Williams, Software performance antipatterns, Proceedings of the second international workshop on Software and performance - WOSP '00 (2000) 127–136doi:10.1145/350391.350420.
URL http://portal.acm.org/citation.cfm?doid=350391.350420

[20] V. Vetoio, PhD Thesis in Computer Science Automated generation of architectural feedback from software performance analysis results Catia Trubiani, Language.

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

[22] C. U. Smith, L. G. Williams, More New Software Performance AntiPatterns: EvenMore Ways to Shoot Yourself in the Foot, Computer Measurement Group Conference (2003) 717–725.
URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.4517{\&}rep=rep1{\&}type=pdf

[23] D. Arcelli, V. Cortellessa, C. Trubiani, Antipattern-Based Model Refactoring for Software Performance Improvement, Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures (QoSA '12) (2012) 33–42doi:10.1145/2304696.2304704.
URL http://doi.acm.org/10.1145/2304696.2304704
[24] T. Mackinnon, S. Freeman, P. Craig, Endo-Testing : Unit Testing with Mock Objects, Extreme programming examined (2001) 287–301.
URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.23.3214{\&}rep=rep1{\&}type=pdf
[25] M. a. Brown, E. Tapolcsanyi, Mock Object Patterns, Matrix (2003) 1–17.
[26] E. A. Hunt, D. Thomas, I. T. Pragmatic, A. Hunt, T. Mackinnon, S. Freeman, Software Construction, Ieee Software (June) (2002) 22–24. doi:10.1109/MS.2004.1259177.
[27] M. Harman, Y. Jia, Y. Zhang, Achievements , open problems and challenges for search based software testing, 8th IEEE International Conference on Software Testing, Verification and Validation (ICST) (Icst).
URL http://www0.cs.ucl.ac.uk/staff/mharman/icst15.pdf
[28] A. Aleti, I. Moser, L. Grunske, Analysing the fitness landscape of search-based software testing problems, Automated Software Engineering (2016) 1–19doi:10.1007/s10515-016-0197-7.
[29] A systematic review of search-based testing for non-functional system properties, Information and Software Technology 51 (6) (2009) 957–976. doi:10.1016/j.infsof.2008.12.005.
[30] V. Garousi, Traffic-aware Stress Testing of Distributed Real-Time Systems based on UML Models using Genetic Algorithms (August).
[31] V. Garousi, A Genetic Algorithm-Based Stress Test Requirements Generator Tool and Its Empirical Evaluation, IEEE Transactions on Software Engineering 36 (6) (2010) 778–797. doi:10.1109/TSE.2010.5.
[32] S. Di Alesio, S. Nejati, L. Briand, A. Gotlieb, Stress testing of task deadlines: A constraint programming approach, IEEE Xplore (2013) 158–167doi:10.1109/ISSRE.2013.6698915.
[33] S. Di Alesio, S. Nejati, L. Briand, A. Gotlieb, Worst-Case Scheduling of Software Tasks – A Constraint Optimization Model to Support Performance Testing, Principles and Practice of Constraint Programming 813–830doi:10.1007/978-3-319-10428-7_58.
[34] S. D. I. Alesio, L. C. Briand, S. Nejati, A. Gotlieb, Combining Genetic Algorithms and Constraint Programming, ACM Transactions on Software Engineering and Methodology 25 (1).
[35] N. J. Tracey, A search-based automated test-data generation framework for safety-critical software, Ph.D. thesis, Citeseer (2000).
[36] J. T. J. Alander, T. Mantere, P. Turunen, Genetic Algorithm Based Software Testing, in: Neural Nets and Genetic Algorithms, 1998.
[37] J. Wegener, H. Sthamer, B. F. Jones, D. E. Eyres, Testing real-time systems using genetic algorithms, Software Quality Journal 6 (2) (1997) 127–135. doi:10.1023/A:1018551716639.
URL http://www.springerlink.com/index/uh26067rt3516765.pdf
[38] B. J. 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.
[39] L. C. Briand, Y. Labiche, M. Shousha, Stress testing real-time systems with genetic algorithms, Proceedings of the 2005 conference on Genetic and evolutionary computation - GECCO '05 (2005) 1021doi:10.1145/1068009.1068183.
[40] G. Canfora, M. D. Penta, R. Esposito, M. L. Villani, 2005., Canfora, G., An approach for QoS-aware service composition based on genetic algorithms.
[41] J. Wegener, M. Grochtmann, Verifying timing constraints of real-time systems by means of evolutionary testing, Real-Time Systems 15 (3) (1998) 275–298. doi:Doi 10.1023/A:1008096431840.
[42] F. Mueller, 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)doi:10.1109/RTTAS.1998.683198.
[43] P. Puschner, R. Nossal, Testing the results of static worst-case execution-time analysis, Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)doi:10.1109/REAL.1998.739738.
[44] H. 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).
[45] H. Gross, B. F. Jones, D. E. Eyres, Structural performance measure of evolutionary testing applied to worst-case timing of real-time systems, Software, IEE Proceedings- 147 (2) (2000) 25–30. doi:10.1049/ip-sen.
[46] M. D. Penta, G. Canfora, G. Esposito, Search-based testing of service level agreements, in: Proceedings of the 9th annual conference on Genetic and evolutionary computation, 2007, pp. 1090–1097.
[47] V. Garousi, Empirical analysis of a genetic algorithm-based stress test technique, Proceedings of the 10th annual conference on Genetic and evolutionary computation - GECCO '08 (2008) 1743doi:10.1145/1389095.1389433.
[48] N. J. Tracey, J. a. Clark, K. C. Mander, Automated Programme Flaw Finding using Simulated Annealing.
[49] H. Pohlheim, M. Conrad, A. Griep, Evolutionary Safety Testing of Embedded Control Software by Automatically Generating Compact Test Data Sequences, Analysis (724) (2005) 804—-814. doi:10.4271/2005-01-0750.
[50] G. R. Raidl, J. Puchinger, C. Blum, Metaheuristic hybrids, in: Handbook of metaheuristics, Springer, 2010, pp. 469–496.
[51] C. Blum, A. Roli, Metaheuristics in combinatorial optimization: overview and conceptual comparison, ACM Computing Surveys 35 (3) (2003) 189–213. doi:10.1007/s10479-005-3971-7.
[52] W. Jaziri, Local Search Techniques: Focus on Tabu Search, 2008.
[53] F. Glover, R. Martí, Tabu Search, Tabu Search (1986) 1–16.
[54] E.-G. Talbi, Metaheuristics: from design to implementation, Vol. 74, John Wiley & Sons, 2009.
[55] J.-Y. Gendreau, Michel and Potvin, Handbook of Metaheuristics, Vol. 157, 2010. doi:10.1007/978-1-4614-1900-6.
[56] Model-based generation of testbeds for web services, Testing of Software and ... (2008) 266–282.