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

Nauber Gois, Pedro Porfírio, André Coelho
Universidade de Fortaleza, UNIFOR, Av. Washington Soares, 1321 , Fortaleza - CE, Brazil
E-mail: naubergois@gmail.com

Abstract

Metaheuristic search techniques have been extensively used to provide solutions for a more cost-effective testing process. 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. Experiments involving stress 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. A Testbed makes possible follow a formalized it's a methodology to reproduce tests for further analysis and comparison. 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 testbed tool named IAdapter TestBed to evaluate various diversity combining metaheuristics in search-based software testing. Two experiments were conducted to validate the proposed tool. In the first experiment the metaheuristics converged to scenarios with no antipatterns. In the second experiment, the metaheuristics excluding the scenarios with Unbalanced Processing antipattern.

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].

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].

Stress 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].

Stress Search-based testing is seen as a promising approach to verifying timing constraints [5]. A common objective of a stress search-based test is to find scenarios that produce execution times that violate the specified timing constraints [7].

Experiments involving stress 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. 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. 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 a testbed[8].

This paper addresses the problem of comparing the use of several metaheuristics in search based tests. In this paper, we propose a flexible testbed tool to evaluate various diversity combining metaheuristics in search based software testing. A tool named IAdapter (github.com/naubergois/newiadapter), a JMeter plugin for performing search-based load tests, was extended [9]. The

DOI reference number: 10.18293/SEKE2015-205

IAdapter Testbed is an open-source tool that provides tools for search based test research. This 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 tool. The experiments uses genetic, algorithms, tabu search, simulated annealing and an hybrid approach proposed by Gois et al. [9].

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 Background

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 [10] [3] [11].

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 [**?**].

Search–Based Testing uses metaheuristic algorithms to automate the generation of test inputs. Metaheuristics are strategies that guide the search process to efficiently explore the search space in order to find optimal solutions [12]. Metaheuristics can be classified as population-based or single solution metaheuristics. 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.

A common goal of load, performance or stress search-based testing is to find workloads that produce execution times that exceed the timing constraints specified. If a temporal error is found, the test was successful [7]. The application of evolutionary algorithms to stress tests involves finding the best- and worst-case execution times (B/WCET) to determine whether timing constraints are fulfilled [5].

IAdapter is a JMeter plugin designed to perform search-based stress tests. The plugin uses genetic algorithms, tabu search and simulated annealing in a collaborative mode (hybrid metaheuristic) [9]. JMeter is a desktop application designed to test and measure the performance and functional behavior of applications.

Many times, a unit test should test a class in isolation. The JMeter tool can test a class in isolation using Mock objects. 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 [**?**] [**?**].

3 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. 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 [13]. The table 1 present some of the most common performance antipatterns.

Blob antipattern is known by various names, including the "god" class [8] and the "blob" [2]. Blob is an antipattern whose problem is on the excessive message traffic generated by a single class or component, a particular resource does the majority of the work in a software. The Blob antipattern occurs when a single class or component either performs all of the work of an application or holds all of the application's data. Either manifestation results in excessive message traffic that can degrade performance [14] [15]. Figure 1 presents a sample where the Blob class uses the features A,B,C,D,E,F and G of the hypotetical system [16][17].

Unbalanced Processing it's characterises for one sce-

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	

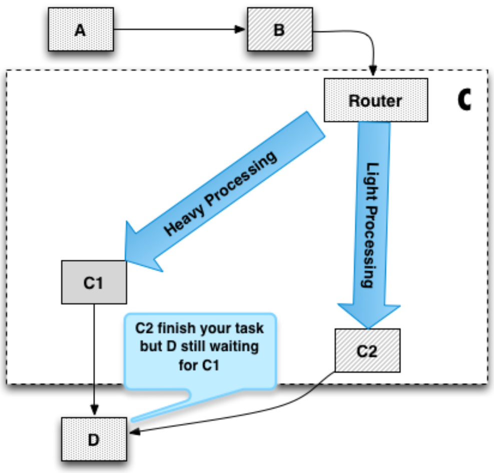

Figure 2: Unbalanced Processing sample [17].

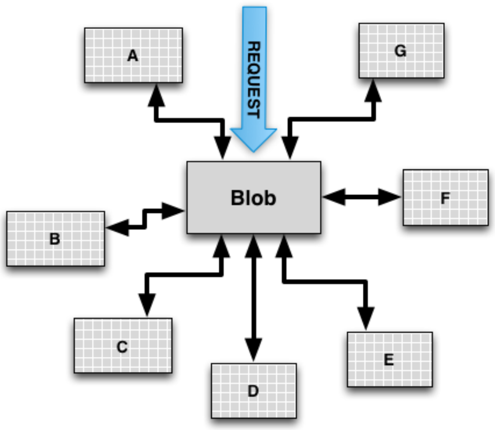

Figure 1: The God class[17].

nario 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. 2 shows a sample of the Unbalanced Processing. In The Fig. 2, four tasks are performed. The task D it is waiting for the task C conclusion that are submmited to a heavy processing situation.

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 [18] [19]. 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 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 [19].

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] [19]. Fig. 3 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.

To emulate the presented antipatterns the testbed solu-

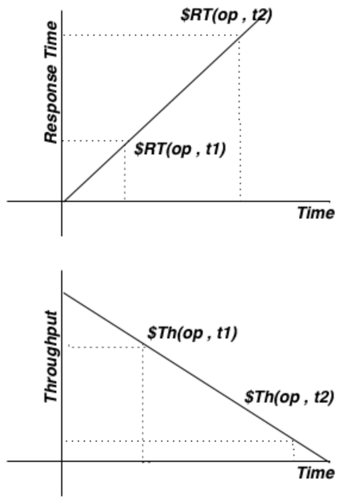

Figure 3: The Ramp sample [16].

tion uses Mock Objects with the JMeter load test tool.

4 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 [9]. 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.

The testbed tool proposed consists of four main elements. Figure 4 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 library 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.

4.1 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. 6 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

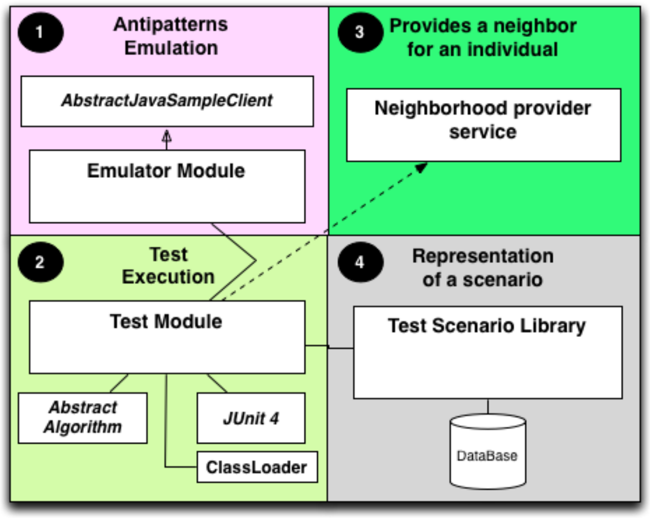

Figure 4: Testbed main architecture.

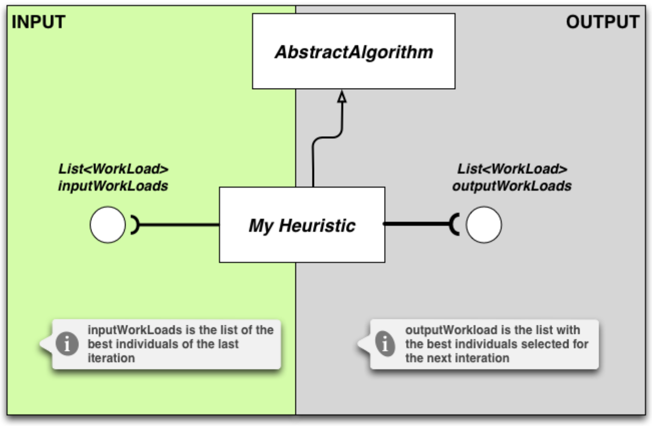
INPUT
OUTPUT
AbstractAlgorithm
List<WorkLoad> inputWorkLoads
List<WorkLoad> outputWorkLoads
My Heuristic
inputWorkLoads is the list of the best individuals of the last iteration
outputWorkload is the list with the best individuals selected for the next interation

Figure 5: Test Module class diagram.

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. Each Metaheuristic could define your own objective function. After all these steps the cycle begins until the maximum number of generations it is reached.

The Fig. 5 shows the class diagram for custom and provided heuristics. All heuristic classes extends the class AbstractAlgorithm. The heuristics receives as input a list of workloads and a list of testcases. Each workload represent an individual in the search space. Each metaheuristic class returns a list of workloads (the individuals selected to the next generation).

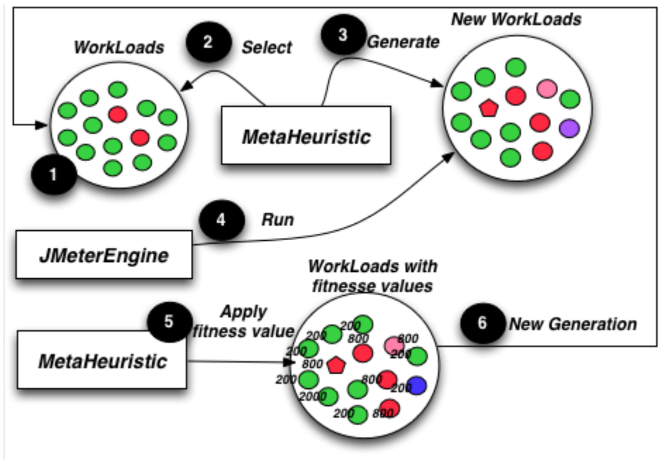

Figure 6: Test Module life cycle.

4.2 Emulator Module

The Emulator Module is responsible for implement and provide successful scenarios and the most commons performance antipatterns (Figure 4 -❶). 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, an Apache JMeter module responsible for unit tests. The Fig. 7 presents the main features of the emulator module. The module implements 8 test scenarios in its first version.

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.

4.3 Test Scenario Library

This modules provides a common representation. The representation for all scenarios. Each 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. 8 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.

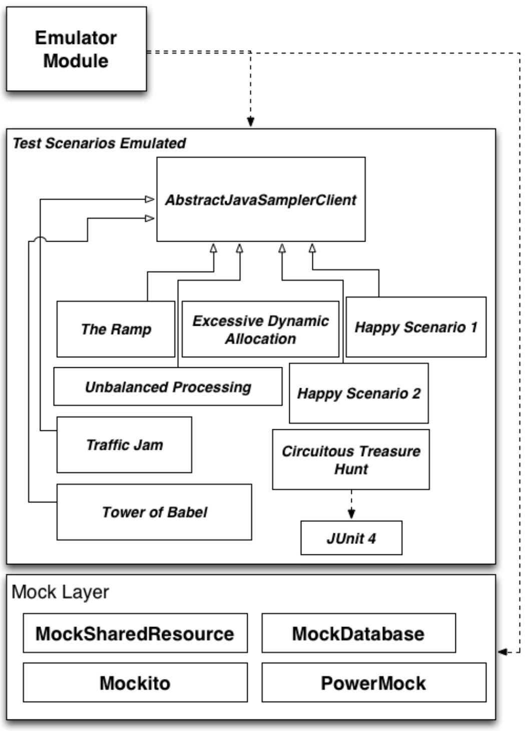

Figure 7: Emulator module

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.

4.4 Neighborhood provider service

Fig. 9 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.

5 Experiments

In this section, We present the results of experiments which we carried out to verify the antipatterns implementation and the metaheuristics used by the testbed tool. We conducted two experiments in order to verify the effectiveness of the testbed tool.

The scope of the experiments is analyze the use of the IAdapter testbed for the purpose of evaluation with respect to effectiveness and efficient from the point of view of the tester in the context of stress test practices. 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 popula-

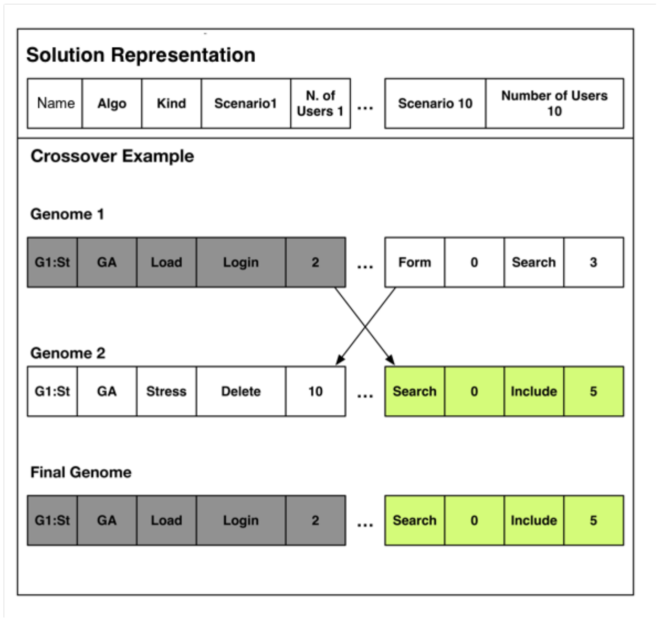

Figure 8: Solution representation and crossover example

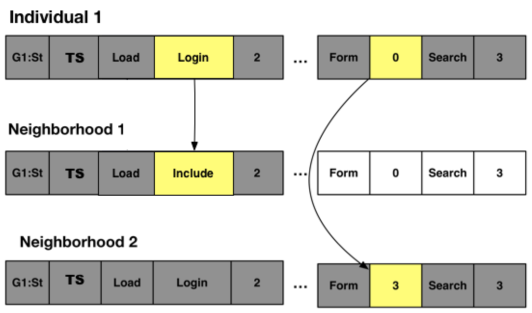

Figure 9: Neighborhood provider strategy

tion on each generation. The experiments uses tabu search, genetic algorithms and the hybrid metaheuristic approach proposed by Gois et al. [9]. 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 expected.

5.1 Research Questions

The following research question is addressed:

- Does the IAdapter testbed reproduce a workload by using an antipattern implementation?
- Does the IAdapter testbed be able to provide client and server metrics with the aim of being used for web performance evaluation studies?
- The IAdapter testbed should be extensible, allowing create new test scenarios?

5.2 Variables

The independent variables are the test scenarios (antipatterns and happy scenarios). The dependent variable are the maximal number of users supported by the application.

5.3 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. 10 and 11 presents the fitness value obtained by each metaheuristic. The SA algorithm obtained the worst fitness values. Hybrid metaheuristic obtained the better fitness values.

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. 12). The Fig. 13 shows the average, minimal e maximum value by search method.

The Fig. 14 presents the maximum, average, median and minimum fitness value by generation. The maximun fitness value increases at each generation. The Fig. 15 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.

Table 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.

Table 2: Best individuals found in the first experiment

Search Method	Generation	Users	fitness Value	Happy 2	Ha
Hybrid	17	145	432760	64	81
Hybrid	17	145	432740	46	99
Hybrid	17	146	431760	54	92
Hybrid	16	143	426740	30	11

Fig. 16 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. 17 presents the response time by number of users of individuals with the Ramp and Circuitous Treasure antipatterns scenarios. The Figure illustrates the smallest number

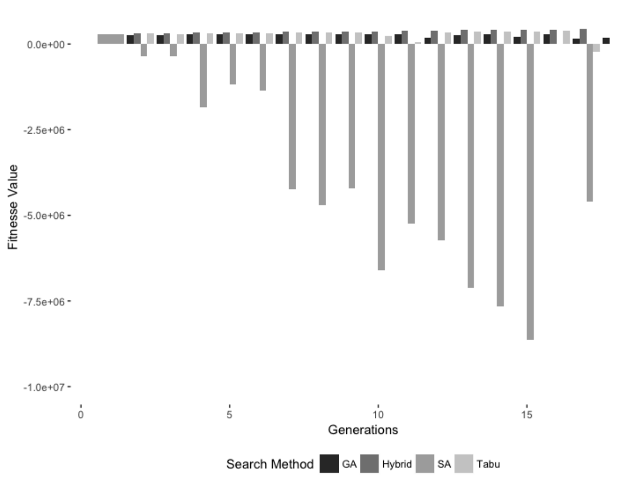

Figure 10: fitness value obtained by Search Method

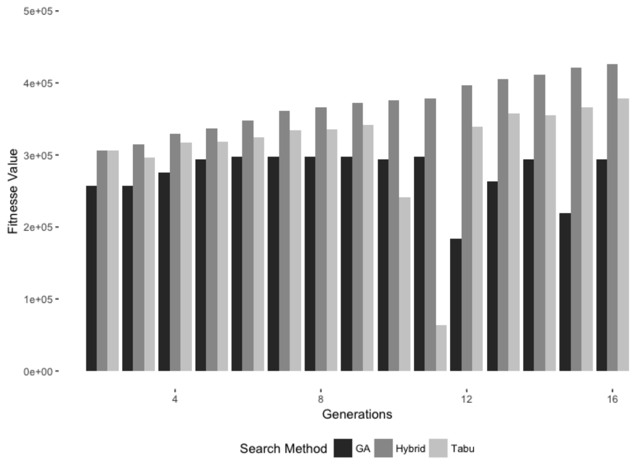

Figure 11: fitness value obtained by Search Method without SA metaheuristic.

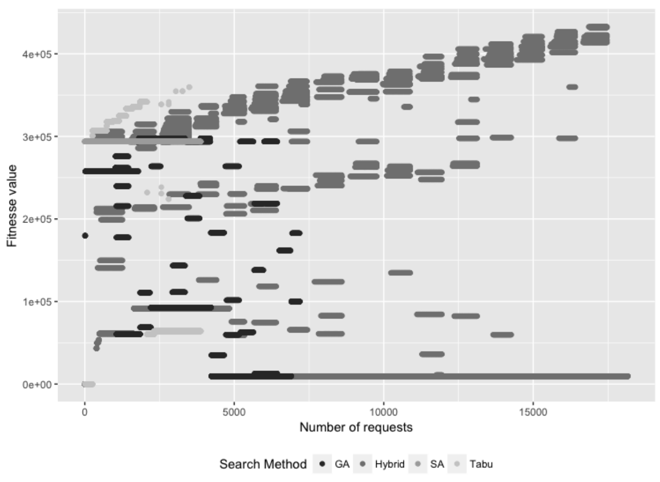

Figure 12: Number of requests by Search Method

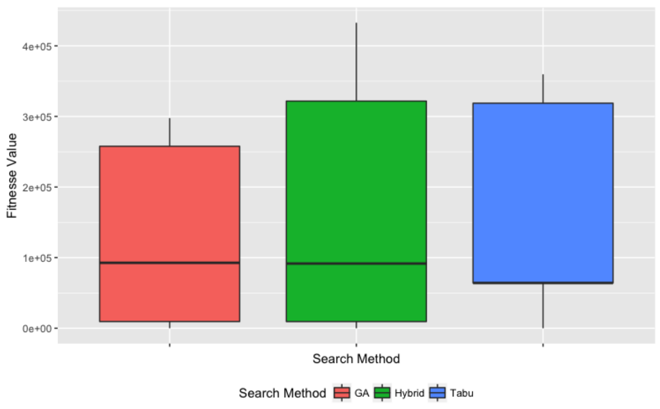

Figure 13: Average, median, maximum and minimal fitness value by Search Method

of individuals with the antipatterns when compared to individuals who use the happy scenarios.

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.

5.4 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. 18 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. 19). The Fig. 20 shows the average, minimal e maximum value by search method. The Fig. 21 presents the maximum, average, median and minimum fitness value by generation. The maximun fitness value increases at each generation. The Fig. 22 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.

Table 3 shows 4 individuals with 145 to 148 users. The first individual has 72 users on Happy Scenario 2, 30 users

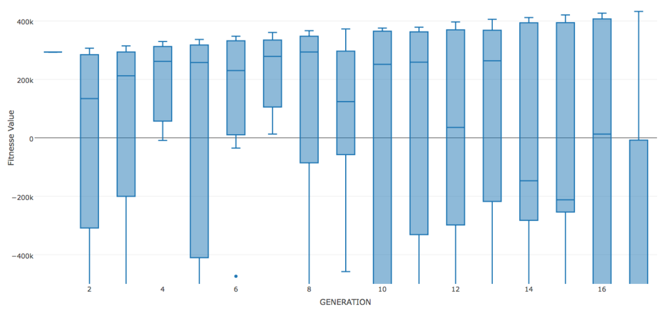

Figure 14: fitness value by generation

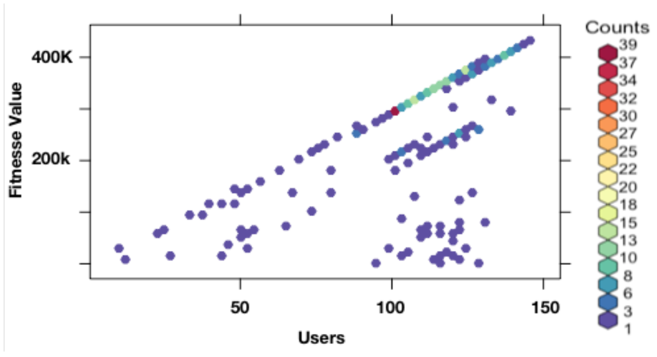

Figure 15: Density graph of number of users by fitness value

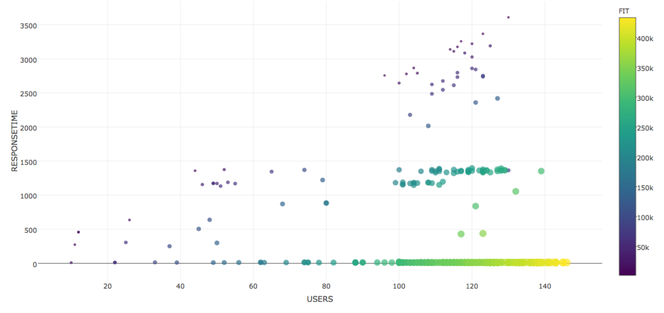

Figure 16: Response time by number of users of individuals with Happy Scenario 1 and Happy Scenario 2

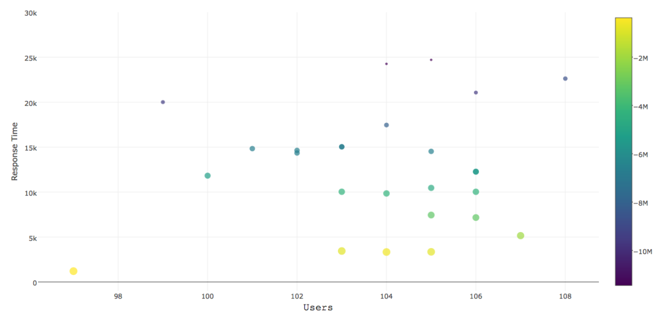

Figure 17: Response time by number of users of individuals with the Ramp and Circuitous Treasure antipatterns

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 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. 23 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. 24 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.

Table 3: Best individuals found in the second experiment

Search Method	Generation	Users	fitness Value	Happy 2	To
Hybrid	17	148	437780	72	46
Hybrid	17	145	432740	71	15
Hybrid	16	146	431800	72	31
Hybrid	17	145	428780	71	32

6 Conclusion

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. 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. [9].

The experiments ran for 17 generations. The experiments used an initial population of 4 individuals by metaheuristic. All tests in the experiment were conducted with-

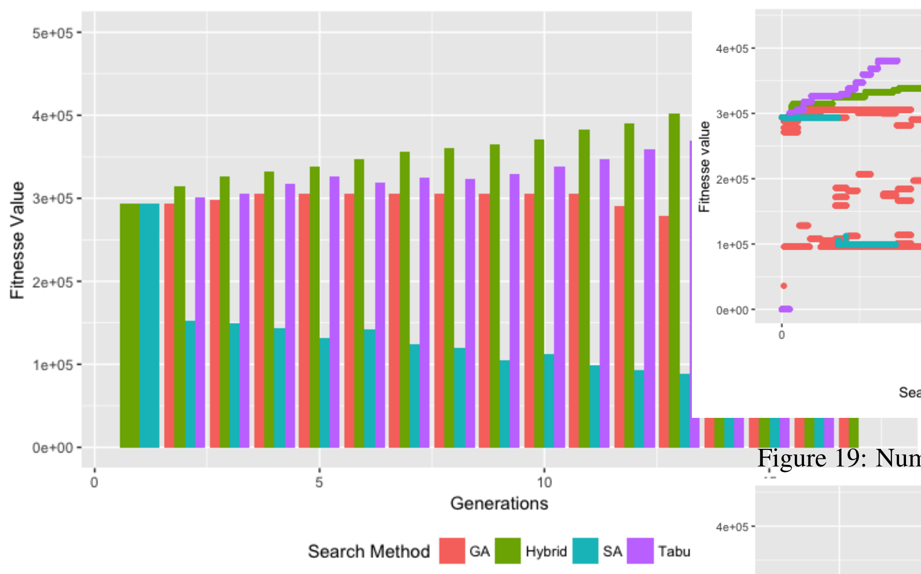

Figure 18: itnesse value obtained by Search Method

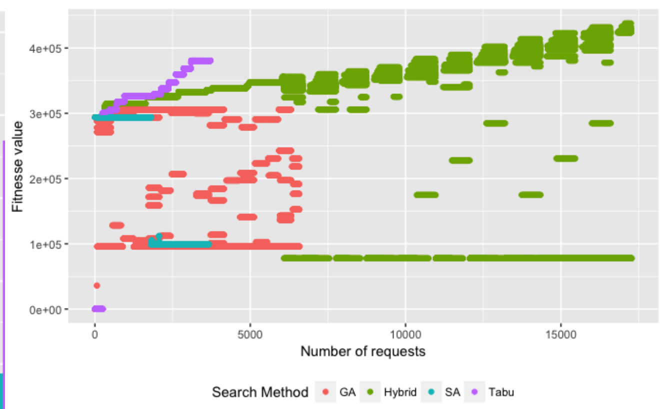

Figure 19: Number of requests by Search Method

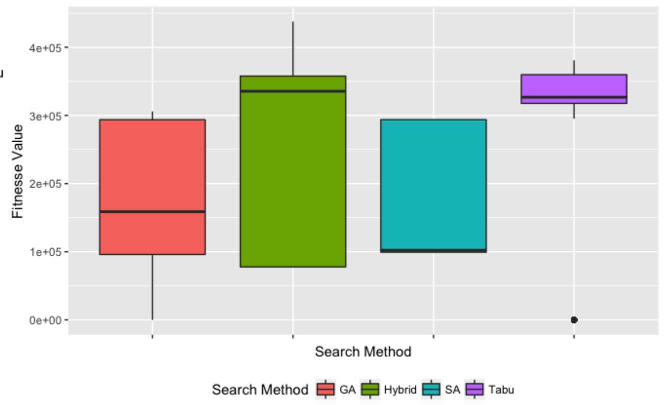

Figure 20: Finesse value by generation in all tests

out 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 first 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 second 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.

References

[1] 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.

[2] Z. Jiang, "Automated analysis of load testing results," Ph.D. dissertation, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1831726

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

[4] A. Wert, M. Oehler, C. Heger, and 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)*, pp. 3–12, 2014. [Online]. Available: http://dx.doi.org/10.1145/2602576.2602579

[5] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, 2009.

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

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

[8] J.-Y. Gendreau, Michel and Potvin, *Handbook of Metaheuristics*, 2010, vol. 157.

[9] 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*

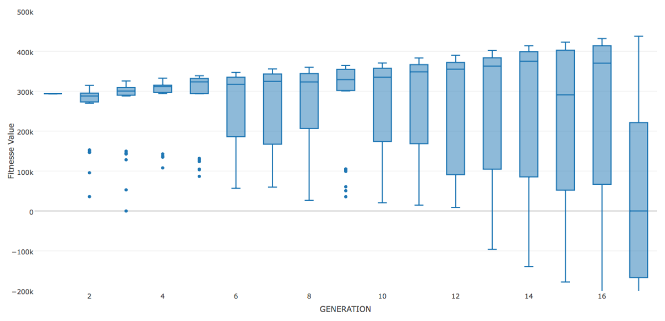

Figure 21: Response time by generation in all tests scenarios

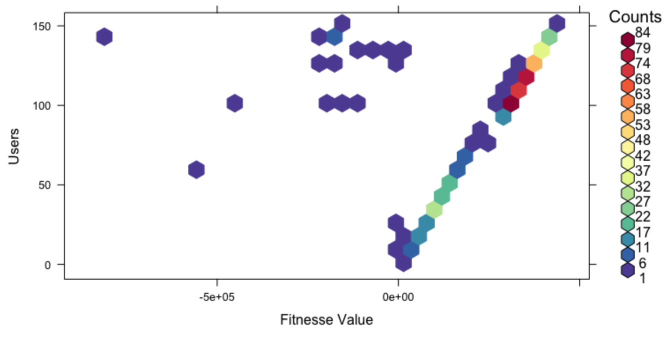

Figure 22: Density graph of number of users by fitness value

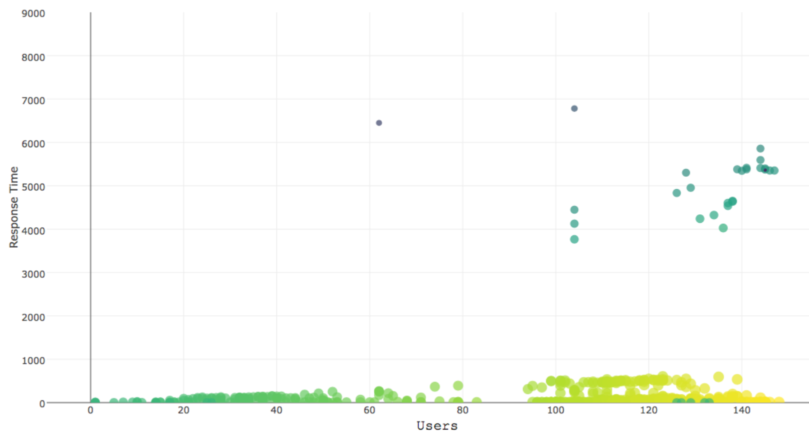

Figure 23: Response time by number of users of individuals with Happy Scenario 1, Happy Scenario 2 and Tower Babel antipattern

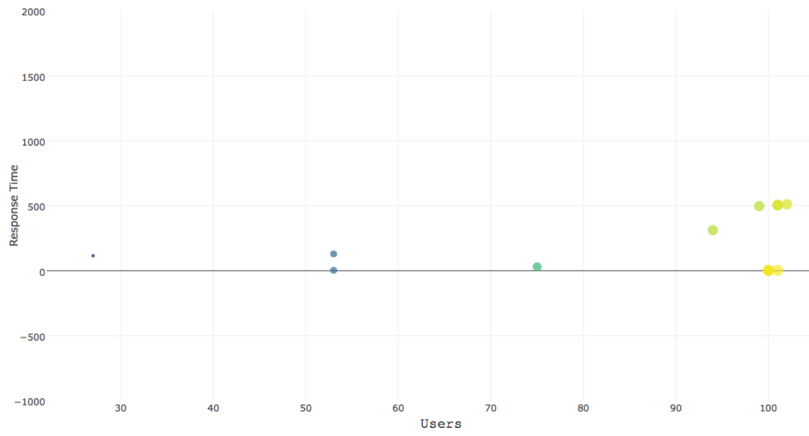

Figure 24: Response time by number of users of individuals with Unbalanced Processing antipattern

(CLEI), 2016, pp. 718–728.

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

[11] 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," 2014.

[12] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: overview and conceptual comparison," *ACM Computing Surveys*, vol. 35, no. 3, pp. 189–213, 2003.

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

[14] V. Cortellessa and L. Frittella, "A Framework for Automated Generation of Architectural Feedback from Software Performance Analysis," pp. 171–185, 2007.

[15] C. U. Smith and L. G. Williams, "Software performance antipatterns," *Proceedings of the second international workshop on Software and performance - WOSP '00*, pp. 127–136, 2000. [Online]. Available: http://portal.acm.org/citation.cfm?doid=350391.350420

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

[17] A. Wert, J. Happe, and L. Happe, "Supporting swift reaction: Automatically uncovering performance problems by systematic experiments," *Proceedings - International Conference on Software Engineering*, no. May, pp. 552–561, 2013.

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

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