Evolutionary Testing of PHP Web Applications with WETT

Francesco Bolis, Angelo Gargantini, Marco Guarnieri, and Eros Magri

Dip. di Ing. dell'Informazione e Metodi Matematici, Università di Bergamo, Italy
{francesco.bolis,angelo.gargantini,marco.guarnieri,eros.magri}@unibg.it

Abstract. One of the current core requirements of web applications is the continuity of the service, because loss in availability can lead to severe economic losses. This is the main reason behind the growing interest in web application testing that offers to researchers several challenges, due to the peculiar nature of these applications. Several classical testing techniques have been extended to deal with web testing. In this paper we propose to extend to web application testing a recent search-based approach that optimizes the generation of the whole test suite. This approach has several advantages over common approaches that optimize the generation of a single test case at a time. We show the technological challenges we have had to face, the architecture of the tool WETT we have developed, and some preliminary results of the experiments.

1 Introduction

The wide diffusion of Internet combined with mobile technologies has produced a significant growth in the demand of web applications with more and more strict requirements of reliability, usability, inter-operability and security [4]. Due to market pressure and very short time-to-market, the testing of web applications is often neglected by developers, although several works, such as [9], analyze the high costs of unavailability of web applications.

To address this problem, the testing community has tried to extend *traditional* testing methods in order to make them suitable for testing web applications. However, traditional testing theories, methods, and tools cannot be used in most cases just as they are, because of the peculiarities and complexities of web applications. For instance, web applications are almost always connected to databases and they are distributed with a client part (often a simple web browser) and a server part (a web server). The diversity of technologies and programming languages involved in the development of modern web applications represents a serious problem for traditional testing techniques. For instance unit tests may have limited efficacy, and thus acceptance testing techniques are preferred, since they try to capture the behavior of the entire web application [2]. However, the automated generation of oracles for acceptance testing is a hard task, because it requires a formal model from which the oracles can be extracted, and thus the manual definition of them is yet a concrete alternative. Due to this fact the test suite must remain of manageable size to avoid the burden of introducing oracles in large test suites.

For these reasons, web application testing still offers many open research issues and challenges. One possible way to deal with it, is to adapt *search-based* techniques for test generation [8]. Note that there exist several approaches for web testing [2,4], however, only a few of them attempt to extend search-based approaches to web application [1,7]. In our paper we try to apply the approach presented in [6,5] and implemented in a tool called EVOSUITE, that generates and optimizes whole test suites towards satisfying a coverage criterion. That approach improves over the state of the art in search-based testing by keeping the size of the test suite under control and by maximizing the coverage of the whole test suite instead of that of single tests.

An evolutionary approach for test generation of web applications has already been presented in [1]. Whilst we share with it several concepts, our approach is different also because we do not assume any automated oracle. This has a strong impact in the design choices we have made, including the following ones.

(1) Our tests are described in terms of user actions in a high level language (presented in Section 2) and not like [1] as arrays of parameters. Tests preserve an intelligible meaning and the user can better complete the oracle part and reproduce the actual scenario that causes a possible bug. (2) The size and the length of our tests is constantly under control. For example, in [1] test suites had around 160 tests, while our tests for the same case study contains only around 10 tests. (3) Our tool evolves the whole test suite and not single test cases, so coverage is maximized but overlapping in coverage among tests is also minimized.

Also Marchetto and Tonella [7] use a search-based approach in order to test Ajax web applications. A search-based algorithm is used for the exploration of the state space. Their approach shares with ours the goal of making the generated tests more efficient, i.e. of increasing the coverage with respect to the number of events in the tests. To this goal, they use a hill climbing algorithm in order to create test suites that maximize diversity.

In this paper we explain the basic algorithms and the architecture of the Web application Evolutionary Testing Tool - WETT- we have developed. We report also some initial experimental results.

2 Our Approach

Given the fact that an important part of web application behavior, and thus code, is tightly related to the interaction with the users, we have chosen to define our test suites in terms of the events that can be executed on the web application interface, and thus, in order to automate the testing phase, we have chosen to use Sahi[1], a *capture and replay tool* that lets users express test cases using scripts and then execute them. Each test case is a list of Sahi statements, chosen among the ones presented in Figure 1. We choose a random test suite TS as initial population. We call $|t|$ the length of the test case t (i.e. the number of statements in t), AUT the application under test, and we have defined a threshold k that represents the maximum length of a test case.

[1] Sahi Web Automation and Testing Tool - http://sahi.co.in/

```
_navigateTo()
_setValue(textbox())
_setValue(textarea())
_setSelected()
_click(radio())
_click(link())
_click(_image())
_click(_imageSubmit())
_click(_submit())
_click(_reset())
```

Fig. 1: Sahi statements

The genetic algorithm used in our application is shown in Algorithm 1. The algorithm is adapted from [5]. It takes as input the initial test suite TS, the maximum number of iterations it, the fitness threshold ft, and the probability of a crossover cp. It evolves the initial test suite and returns as output the evolved test suite *population*. The algorithm evolves the test suite until the iteration threshold is reached or an adequate test suite is found. At each iteration the algorithm selects the subset of the current population with the best values of fitness using the function $elite(population)$, then it evolves the current elite. It selects two individuals using tournament selection, with the function $tournament(population)$, and it modifies them using the crossover operator with a probability cp. Then the two individuals are mutated and the algorithm adds the two parents or the two new individuals to the current elite, depending on which are the best ones in terms of the fitness. The algorithm continues to evolve the elite until the test suite has grown more than the current population, then it starts another iteration considering the current elite as the population.

We have defined the fitness function in terms of the statement coverage achieved by the test suite. However, given the fact that a web application consists of several pages, in our fitness function we consider also the number of pages covered by the test suite, i.e. we increase the fitness function of a certain value $(n - l)r$ if the number n of covered pages is greater than a certain threshold l, where r represents the reward per page. In this way, we can reward with higher fitness values those test suites that exercise the AUT both in terms of covered statements and in terms of visited pages. Explicitely, the fitness achieved by the test suite T is $f(T) = \alpha * C(T) + \beta * r * (max(n, l) - l)$, where α and β are two parameters used to weight the influence of the statement coverage and the page coverage on the overall fitness value.

Our approach uses the *two points crossover* technique, i.e. given two individuals P_1 and P_2, we generate two new individuals O_1 and O_2 such that O_1 contains the firsts x statements of P_1, then the next $y - x$ statements of P_2, and the lasts $|P_1| - y$ statements of P_1 and the opposite for P_2, where $x, y \in [0, min(|P_1|, |P_2|)]$ are random values such that $x < y$.

In the mutation phase we apply three operators, each one applied with probability $\frac{1}{3}$: (a) *Remove*: Given a test case t, each statement is removed with a probability $\frac{1}{|t|}$, (b) *Change*: Given a test case t, each statement is modified with a probability $\frac{1}{|t|}$ (we change the parameters of the statement by choosing new parameters at random from configuration files), (c) *Insert*: We add a new statement to the test case t with a probability β^2, when a new statement is added we add another statement with probability β^{i2} (where i is the number of already inserted statements), until no new statements are added or the maximum length of the test case is reached. The statements to add are selected randomly among the ones presented in Listing 1, and the parameters are chosen at random from

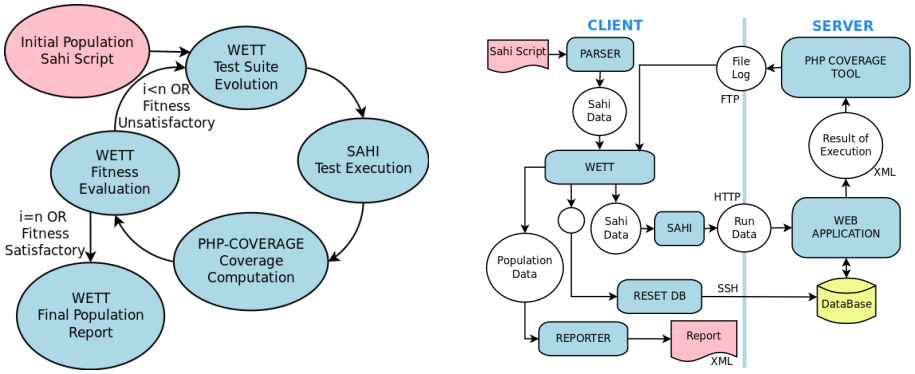
Algorithm 1: Genetic Algorithm

```
Input  : ts, it, ft, cp
Output: population
begin
    population ← TS;
    iteration ← 0;
    while (iteration < it) ∧ (fitness(population) < ft) do
        E ← elite(population);
        while |E| < |population| do
            P1, P2 ← tournament(population);
            if random([0, 1]) < cp then O1, O2 ← crossover(P1, P2);
            else O1, O2 ← P1, P2;
            mutate(O1, O2);
            fp ← max(fitness(P1), fitness(P2));
            fo ← max(fitness(O1), fitness(O2));
            Tb ← best individual of population;
            if fp > fo then E ← E ⋃ {P1, P2};
            else
                for O ∈ {O1, O2} do
                    if length(O) ≤ 2 * length(Tb) then E ← E ⋃ {O};
                    else E ← E ⋃ {P1 or P2};
        population ← E;
```

the configuration files. Configuration files contain the definitions of the elements of the AUT that can be used as arguments for the Sahi statements.

In order to prevent the undefined growth in length of the test cases, which could cause a *bloat*, we adopted several strategies: (i) we have defined a maximum length for the test cases, (ii) we have used the *tournament selection* method, that lets us choose only the individuals with high fitness values, (iii) our fitness function considers also the number of pages explored and, thus, we can reward better the individuals that explore several pages instead of the ones that explore exhaustively a single page.

2.1 Architecture



(a) Process. (b) Architecture.

Fig. 2: Our Approach

As already said, each WETT test case is a list of Sahi statements, which are executed by the Sahi tool that invokes the browser and executes the scripts. Figure 2(a) presents our process, whereas Figure 2(b) shows the architecture of the WETT tool. In order to generate the initial population, we execute manually a certain number of test cases using Sahi, and then we select at random the population among the statements used in these test cases. Then we parse the scripts in order to import them into the WETT application, where we evolve the current population. In order to compute the fitness function, we execute the evolved test suite by using Sahi, which communicates with the AUT using the HTTP protocol. We use *XDebug*[2] and *PHP-Coverage*[3], which are installed on the server that executes the AUT to measure the coverage achieved by the test suite. Given the fact that these tools produce coverage reports on the server, we import the coverage results in WETT via FTP. When WETT computes a test suite that satisfies the fitness requirement or it reaches the iteration threshold, we produce the final test suite and an XML report containing the information about the process. Given the fact that usually web applications use databases to store the current state of the application, we delete the content of the database after the execution of each iteration by executing an appropriate script (RESET DB) by SSH.

3 Initial Experimental Results

In order to evaluate the performance of WETT, we have selected as case study the *Schoolmate*[4] PHP application, which was already used in other works [1,3]. We report here the results of our preliminary experiments. We run WETT five times, each one with a different seed. The use of Sahi implies a considerable time overhead since the execution of a single candidate test case is around one minute. Each experiment started from a test suite with 10 randomly generated test cases and with the elitism that chooses at each iteration the 5 best test cases in terms of fitness. The results in Figure 3 show how the fitness evolves during the iterations. Note that the fitness of the test suite can also decrease from an iteration to the next one due to the elitism that selects each time only a subset of the current population and to the bloat control that limits the length of the test cases.

Table 1 reports the size of the best test suite for every iteration, the coverage achieved by the test suite, and the number of PHP interpreter warnings, which are non-fatal errors. The table reports also the number of visited pages. In comparison with other approaches, our test suite achieves the average statement coverage of 21.7% with 13 test cases, while [1] achieves the 56.5% of branch coverage with 167 test cases and [3] achieves the 64.9% of line coverage with 724 tests. In comparison with a random approach [3], we achieve much better results in terms of coverage (8.3% with random) and test suite size (1396 tests with

[2] XDebug, Debugger and Profiler Tool for PHP - http://xdebug.org/
[3] PHPCoverage, code coverage tool for PHP - http://phpcoverage.sourceforge.net/
[4] Schoolmate - http://sourceforge.net/projects/schoolmate/

random). Note that although our approach achieves a low coverage, it keeps the test suite very small. Our approach visits, in average, 18.4 pages out of 63 pages.

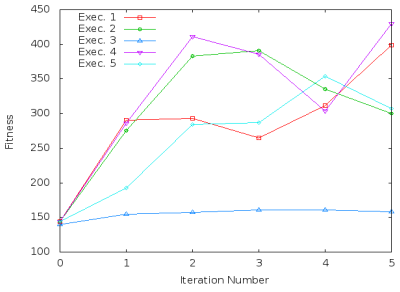


Fig. 3: Fitness Function

#	Test Suite Size	Coverage	Warnings	Visited Pages
1	12	24.94%	36	19
2	17	24.44%	96	20
3	7	10.06%	132	15
4	13	26.87%	72	20
5	13	22.12%	144	18
Avg.	12.4	21.69%	96	18.4

Table 1: Experimental results

4 Conclusion and Future Work

We found that extending the whole test suite generation approach to web application testing is feasible and that our approach has some advantages and drawbacks. We were able to automatically generate small tests suites achieving a discrete coverage. However, the achieved coverage remains low and the required time for test generation makes the number of iteration small. The former drawback is primarily due to the fact that our approach does not consider the state of the web application when mutating test cases and that the initial test suite is randomly chosen: this results in test scripts with a low quality and slows the evolution process. We plan to integrate a model-based/model discovery approach in order to solve this problem. The latter drawback is due to the use of Sahi as capture and replay tool. We plan to consider other alternative tools.

References

1. N. Alshahwan and M. Harman. Automated web application testing using search based software engineering. In *Proc. of ASE 2011*.
2. A. Andrews, J. Offutt, and R. Alexander. Testing web applications by modeling with FSMs. *Software and Systems Modeling*, 2005.
3. S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Trans. Softw. Eng.*, 36(4), 2010.
4. G. Di Lucca and A. Fasolino. Testing web-based applications: The state of the art and future trends. *Information and Software Technology*, 48(12), 2006.
5. G. Fraser and A. Arcuri. Evolutionary generation of whole test suites. In *Proc. of QSIC 2011*.
6. G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proc. of ACM SIGSOFT ESEC/FSE*, page 416–419, 2011.
7. A. Marchetto and P. Tonella. Search-based testing of ajax web applications. In *Proc. of SSBSE 2009*.
8. P. McMinn. Search-based software test data generation: a survey. *Softw. Test, Verif. Reliab*, 14(2), 2004.
9. S. Pertet and P. Narasimhan. Causes of failures in web applications. *CMU TR*, 2005.