GENETIC ALGORITHM FOR TESTING WEB APPLICATIONS

Nashat Mansour, Ramzi Haraty, and Hratch Zeitunlian

Department of Computer Science and Mathematics,
Lebanese American University, Lebanon

ABSTRACT

We present a metaheuristic algorithm for testing software, especially web applications, that can be modelled as a state transition diagram. We formulate the testing problem as an optimization problem and use a genetic algorithm to generate test cases as sequences of events. This algorithm evolves solutions by maximizing a fitness function that is based on testing objectives such as the coverage of events, diversity of events, and continuity of events. The proposed approach includes weights that can be assigned to events. These events would lead to important features or web pages in order to ensure that test cases will be generated to cover these features. The effectiveness of the genetic algorithm is compared with that of other algorithms, namely simulated annealing and a greedy algorithm. Our experimental results show that the proposed genetic algorithm demonstrates serious promise for testing state-based software, especially web applications.

KEYWORDS

Genetic Algorithm, Metaheuristics, Search Based Software Engineering, State-Based Testing, Testing Web Applications

1. INTRODUCTION

Web applications, including Web 2.0, have evolved significantly and are no longer represented as static pages. The web is approached as a platform, and software applications are built upon it [1]. Web 2.0 applications are built around several technologies that can be executed within webpages, etc…. The new technologies introduce additional challenges for testing web application such as those of the dynamic user interface elements and states [2]. Hence, existing web testing methods [3, 4, 5] are not sufficient and new approaches are required.

In order to reduce testing costs and improve software quality, several web application testing tools have been proposed [6]. An Extended Finite State Machine (EFSM) can often be viewed as a compressed notation of an FSM. Petrenko and Boroday [7] call the state of unfolded EFSM as "configuration" and investigate the problem of constructing a configuration of sequences from an EFSM model. Memon and Pollack [8] worked on artificial intelligence planning to manage the state-space explosion by eliminating the need for explicit states. In their work, the GUI description is manually created by a tester in the form of planning operators, which model the preconditions and post-conditions of each GUI event. The planner automatically generates test cases using pairs of initial and destination transitional states. Liu et al. [9] propose a formal technique that models web application components as objects and generates test cases based on data flow between these objects. Ricca and Tonella [10] present a test generation model based on the Unified Modelling Language. These techniques extend traditional path-based test generation and use forms of model-based testing. They can be classified as “white-box” testing techniques since the testing models are generated from the web application code.

Not much research has been reported on testing web applications with dynamic features using state transition diagrams. Marchetto et al. [2] proposed a state-based testing technique designed to address the new features of Web 2.0 applications. In this technique, the DOM manipulated by AJAX code is abstracted into a state model where call-back executions triggered by asynchronous messages received from the web server, are associated with state transitions. The test cases are generated from the state model based on the notion of semantically interacting events. However, this technique generates a very large number of test cases that could limit the usefulness of the test suites. Another proposal by Marchetto et al. [11] proposed a search-based approach based on a hill-climbing algorithm to generate test sequences while keeping the test suite size reasonably small. In order to preserve a fault revealing power comparable to that of exhaustive test suite, they aimed to maximize the diversity of the test cases. The industry also proposed several functional testing tools for testing web application. Some tools rely on discovering and systematically exploring website execution paths that can be followed by a user in a web application [12]. Further approaches to functional testing are based on user session data to produce test suites [13]. Others are based on HttpUnits where the application is divided to HttpUnits and tested by mimicking web browser behaviour [14].

In this paper, we propose an effective state-based testing method, which can be used to handle the complexity of web applications. We model web applications by associating features or web pages with states and events that represent state transitions. Then, metaheuristics, namely a genetic algorithm is designed to generate a controlled number of test cases with maximum diversity and coverage. The genetic algorithm is population-based and evolves results over many generations of candidate solutions using nature-inspired genetic operators. This metaheuristic algorithm has demonstrated its effectiveness for classical software testing [15]. Also, it evolves solutions by minimizing a function that represents the testing objectives. Moreover, although the proposed method is presented for and applied to examples of web applications, it is appropriate for software applications that can be modelled by a state graph.

In the next section, we present the specificities pertaining to testing web applications, including how to build a state-based model for web applications. In section 3, we present the metaheuristic, genetic algorithm. Experimental results are discussed in section 4. Section 5 concludes the paper.

2. Testing Web Applications and State Graph Modeling

Testing is an essential part of the software development cycle. Web applications differ from traditional software development where they follow the agile software development model, which has shorter development time. Because of the short development time, web applications usually lack necessary documents during the development and the user requirements often change. Testing and maintaining web applications becomes a more complex task compared to traditional software. During the past decade radical changes were introduced to the development of web applications and even the concept of the web. The web is approached as a platform where software applications are built upon; thus, the emergence of a new generation of web applications and web systems known as Web 2.0. Web 2.0 applications are based on highly dynamic web pages, build around AJAX technologies, which through the asynchronous server calls, enable the users to interact and affect the business logic on the servers. The dynamic features of Web 2.0 add more complexity to the hard task of testing web application [16].

We propose a state based testing strategy that will dynamically generate a finite state machine from a web application by extracting interacting events [11] that produce state changes in the user interface. From the inferred graph, test cases will be generated as a sequence of events. However, generating test case sequences from the finite state machine can lead to a very large number of test cases in the test suites. This is why Marchetto et al. [11] suggested a search based approach to generate long sequences of events while keeping the test suite size reasonably large using a hill-climbing algorithm. The problem with this algorithm is that the solution will be a local optimum rather than being a global optimum.

The objective of our research is to develop a more effective state based testing for a Web 2.0 application that will cover its dynamic features. This testing approach is based on metaheuristic algorithms rather than exact graph algorithms for traversing the events in the state-based graph model. That is, the objective is to develop optimal or good suboptimal test suite that reduces the number of test cases and not merely a set of sequences of events/edges in the graph. Other considerations can be found in [16].

In addition, extracting a state graph from a Web 2.0 application is not a direct and simple task. The challenges are described in [16]. Our testing mechanism reconstructs the user interface states, and generate static pages having navigation paths each with a unique URL. These static pages will be used to conduct state-based testing [2]. To attain the static-like pages we need a tool that will execute client side code, and identify clickable elements which may change the state HTML/DOM within the browser. From these state changes, we will build our state graph that captures the states of the user interface, and the possible transitions between the states. The definition of the state graph is found in [16]. Furthermore, two issues are to be considered while building the state graph. First, we need to detect the event-driven elements; next, we need to identify the state changes. The state graph is created incrementally; initially, the state graph contains only the root state. Additional states are appended to the graph as event-driven elements are traced/invoked in the application and state changes are analyzed.

3. GENETIC ALGORITHM

Genetic Algorithms [17] simulate the natural phenomenon of populations' reproduction and selection operations in order to achieve optimal results. Through artificial evolution, successive generations search for fitter adaptations. Each generation consists of a population of chromosomes, also called individuals, and each chromosome represents a candidate solution. The Darwinian principle of reproduction and survival of the fittest and the genetic operations of recombination (crossover) and mutation are used to create a new offspring population from the current population. The process is repeated for many generations with the aim of maximizing the fitness of the individuals. In the following subsections, we describe how we generate test sequences of semantically interacting events using the genetic algorithm; an outline of the genetic algorithm is given in Fig. 1.

```
Random generation of initial population, size POP;
Evaluate fitness of individuals;
repeat
Rank individuals and allocate reproduction trials;
for i = 1 to POP step 2
        Randomly select two parents from list of reproduction trials;
                Apply crossover and mutation;
endfor
Evaluate fitness of offspring;
Save_best_so_far();
until  convergence;
```

Figure 1. Outline of the genetic algorithm

3.1. Chromosomal representation and fitness function

GA's population is an array of POP individuals (candidate solutions). An individual in the population is implemented as a vector of variable-length test cases. Each test case is represented by a sequence of a maximum of *K* events derived from the state graph. The vector's length of is *K*N*, where *N* is the maximum number of test cases required in the candidate solution. To allow variable length of test cases, we introduce a random number of fake edges into our set of valid events. These fake edges, called "No Edge", will play the role of space holder in the array.

The fitness function is composed of three weighted factors, continuity, diversity and coverage, which are explained in [16]. Briefly, testing a continuous set of events for event-based applications is likely to reveal faults. We aim to eliminate or minimize the discontinuity (*DC*) of events in a test case. Diversity guarantees that test cases will cover events from the entire scope of the web application. The lack of diversity (*LDiv*) is minimized by calculating the average frequency of events in the entire test suite. In Web 2.0 applications, end users and third parties can change the content of a web page dynamically and some events would have higher importance/weight than others. The weighted coverage (WC) is the sum of weighted coverage of events. The fitness function (to be maximized) is represented as the reciprocal of

$$\mathrm{E} = \alpha \times \frac{1}{WC} + \beta \times LDiv + \gamma \times DC$$

where α*, β, and* γ are user-defined weights which will allow flexibility in using our proposed algorithm to suit the user's particular choices.

3.2. Reproduction scheme and convergence

The whole population is considered a single reproduction unit within which random selection is performed. The reproduction scheme is based on ranking, followed by random selection of mates from the list of reproduction trials assigned to the ranked individuals. In the ranking scheme for the population, the individuals are sorted by fitness values, and ranks are assigned to individuals based on a scale of equidistant values. The ranks assigned to the fittest and least-fit individuals are 1.25 and 0.75, respectively. Individuals with ranks greater than 1 are first assigned single copies. Then, the fractional part of their ranks and the ranks of the lower half of individuals are treated as probabilities for random assignment of copies. Elitism is used to exploit good building blocks and to ensure that good candidate solutions are preserved. This is done by replacing the least-fit individual with the best-so-far individual if the latter is better than the current-fittest. Convergence is detected when the best-so-far candidate solution does not change its fitness value for 10 generations.

3.3. Genetic operators

The genetic operators are 2-point crossover and mutation at the rates 0.7 and 0.01, respectively. To apply the operators, we start with random selection of pairs of chromosomes from the mating pool, at the rate of 0.7. Each pair of these chromosomes undergoes crossover, with the condition that the randomly chosen crossover points will be aligned with the starting positions of the test cases that are represented by a sequence of *K* events (refer to section 4.1). Thus, all genes between the crossover points are swapped to create two new chromosomes. After that, mutation is applied to randomly selected genes/events, at a rate of 0.01, by randomly changing an event within a proposed test case. At this stage, the new population of offspring replaces that of the parents. We note that the genetic operators enhance the exploration feature of the algorithm in the large search space (also the fitness landscape). However, crossover and mutation may produce chromosomes that represent infeasible solutions by violating the definition of continuity. This concern is addressed by the fitness function formula, where the discontinuity term (*DC*) is assigned a large weight so that the infeasibility will be severely penalized and, thus, reduces the likelihood of survival of infeasible chromosomes.

4. EXPERIMENTAL RESULTS

The results of generating test cases using a genetic algorithm are presented in this section. These results are compared with those of simulated annealing and greedy algorithms. The simulated annealing algorithm makes use of the same objective/energy function (*E*). However, at the end of each iteration, the event frequency, coverage, and diversity matrices are saved, to be used by the energy function on the next iteration. The greedy algorithm is designed to accept only the changes that decrease the energy/objective function value, and not to allow any uphill moves. It deals with the entire test suite instead of generating a single test case after each iteration.

The three algorithms are applied on a state graph with 270 events as shown in Fig. 2. A test suite of 40 test cases is developed; each test case having a maximum of *K* test events. To deal with variable-length test case size, we append fake edges ("No Edge") to the list of events. The three algorithms were run with different test case sizes *K* = 10, 15, 20 and the execution of each algorithm was repeated ten times.

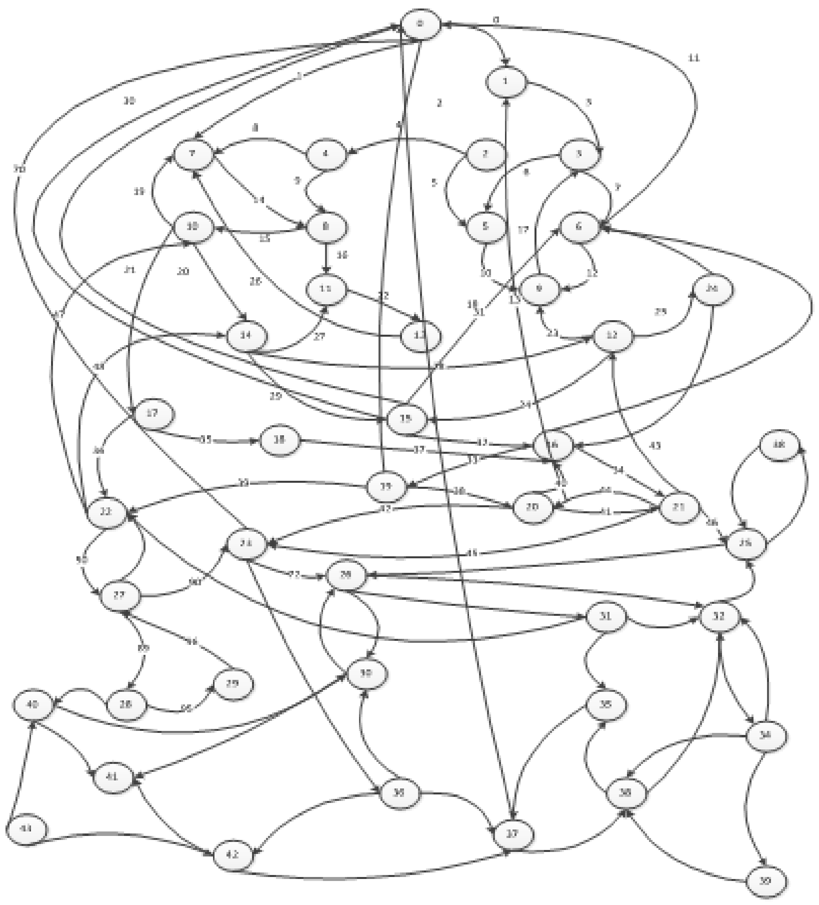

Figure 2. State graph of a web application.

The genetic and annealing algorithms successfully generated 40 useful test cases, consisting of a continuous sequence of events. The results demonstrated that the two algorithms successfully covered all the events in the application. They were also able to generate diverse suite of test cases that efficiently test different parts of the Web 2.0 application. Furthermore, the overall effectiveness of the genetic algorithm was close to that of the simulated annealing algorithm, although the genetic algorithm showed slight advantage. But, the greedy algorithm resulted in an un-optimized test suite. The greedy algorithm failed to generate continuous sequences of events in the test cases. Its objective function converged fast, within the early iterations, and no further improvements were attained. Tables 1-2 show a comparison between the best and average energy function for the three algorithms with different event numbers (maximum K) in a test case. Table 3 presents the standard deviation of the objective function values. It also shows that the genetic and simulated annealing algorithms have reasonable standard deviation, although the genetic algorithm is better on this aspect. But, the greedy algorithm has much wider standard deviation.

Table 1. Best final objective function (*E*) values of 10 runs for different test case sizes.

Max. no. of events in test cases	**Genetic Algorithm**	**Simulated Annealing**	**Greedy Algorithm**
K=10	1.26	1.35	31.00
K=15	1.82	1.81	56.16
K=20	2.73	2.41	83.32

Table 2. Average final objective function (*E*) values of 10 runs for different test case sizes.

Max. no. of events in test cases	**Genetic Algorithm**	**Simulated Annealing**	**Greedy Algorithm**
K=10	1.31	1.77	35.08
K=15	1.87	2.17	61.57
K=20	2.92	2.78	88.66

Table 3. Standard deviation of final objective function (*E*) values of 10 runs for different test case sizes.

Max. no. of events in test cases	**Genetic Algorithm**	**Simulated Annealing**	**Greedy Algorithm**
K=10	0.03	0.19	2.01
K=15	0.03	0.23	3.28
K=20	0.17	0.14	3.73

5. CONCLUSIONS

We designed metaheuristic algorithms for testing web applications. We also modeled the dynamic features of Web 2.0 using state transition diagrams. A genetic algorithm was used to generate test cases. These test cases were generated as sequences of semantically interacting events. We also formulated a fitness function that is based on the criteria of providing high coverage of events, high diversity of events covered, and definite continuity of events. The experimental results show that the genetic algorithm is a promising approach for testing web applications and state-based software.

ACKNOWLEDGEMENTS

This work was partially supported by the Lebanese American University and its contributions were equal among the three authors.

REFERENCES

[1] T. O'Reilly (2013) “Design Patterns and Business Models for the Next Generation of Software,” Retrieved from <http://oreilly.com/web2/archive/what-is-web-20.html> on July 18.

[2] A. Marchetto, P. Tonella, and F. Ricca (2008), “State-based Testing of AJAX Web Applications,” In Proceedings of IEEE International Conference on Software Testing, Lillehammer, Norway, April.

[3] A. Andrews, J. Offutt, and R. Alexander (2005) “Testing Web Applications by Modelling with FSMs,” Software and System Modelling, vol. 4, no. 3, July.

[4] A. Tarhini., N. Mansour. H. Fouchal (2010) “Testing and Regression Testing for Web Services Based Applications,” International Journal of Computing and Information Technology, vol. 2, no. 2, pp. 195 – 217.

[5] G. A. Di Lucca, A. R. Fasolino, F. Faralli, and U. D. Carlini (2002) “Testing Web applications,” In Proceedings of the International Conference on Software Maintenance, Montreal, Canada, October. IEEE Computer Society.

[6] Web Application Testing Tools. Retrieved on July 18, 2013 from <http://logitest.sourceforge.net/logitest/index.html>.

[7] A. Petrenko, S. Boroday and R. Groz (2004) “Confirming Configurations in EFSM Testing,” IEEE Transactions on Software Engineering, vol. 30, pp. 29-42.

[8] M. Memon, M. Pollack and L. Soffa (2001) “Hierarchical GUI Test Case Generation using Automated Planning,” IEEE Transactions on Software Engineering, vol. 27, no. 2, pp. 144–155.

[9] C. Liu, D. Kung, P. Hsia, and C. Hsu (2000) “Structural Testing of Web Applications,” In Proceedings of the 11th IEEE International Symposium on Software Reliability Engineering, pp. 84–96, October.

[10] F. Ricca and P. Tonella (2001) “Analysis and Testing of Web Applications,” In Proceedings of the International Conference on Software Engineering, pp. 25–34, May.

[11] A. Marchetto, P. Tonella, and F. Ricca (2009) “Search-Based Testing of AJAX Web Applications,” In Proceedings of IEEE Search Based Software Engineering, May.

[12] M. Benedikt, J. Freire, and P. Godefroid (2013) “VeriWeb: Automatically Testing Dynamic Web Sites,” Retrieved from <http://www2002.org/CDROM/alternate/654/> on July 18.

[13] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher (2005) “Leveraging User Session Data to Support Web Application Testing,” IEEE Transactions on Software Engineering, vol. 31, no. 3, pp. 187-202.

[14] B. Fejes (2013) “Test Web Applications with HttpUnit,” Retrieved on July 18, from <http://www.javaworld.com/javaworld/jw-04-2004/jw-0419-httpunit.html >.

[15] N. Mansour and M. Salame (2004) “Data Generation for Path Testing,” Software Quality Journal, vol. 12, pp. 121-136.

[16] N. Mansour, H. Zeitunlian, and A. Tarhini (2013) "Optimization Metaheuristic for Software Testing," In: Schütze O. et al. (eds) EVOLVE - A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation II. Advances in Intelligent Systems and Computing, Vol. 175. Springer, Berlin, Heidelberg.

[17] D. Goldberg, Genetic algorithms in search, optimization and machine learning, Addison-Wesley, 1989.

[18] R.A. Rutenbar (1989) "Simulated annealing algorithms: an overview," IEEE Circuits and Devices Magazine, vol. 5, Issue 1.