J. Software Engineering & Applications, **2010, 3, 541-547**
doi:10.4236/jsea.2010.36062 Published Online June 2010 (http://www.SciRP.org/journal/jsea)

Scientific Research

User Session-Based Test Case Generation and Optimization Using Genetic Algorithm*

Zhongsheng Qian

School of Information Technology, Jiangxi University of Finance & Economics, Nanchang, China.
Email: changesme@163.com

Received March 22nd, 2010; revised April 12th, 2010; accepted April 13th, 2010.

ABSTRACT

An approach to generating and optimizing test cases is proposed for Web application testing based on user sessions using genetic algorithm. A large volume of meaningful user sessions are obtained after purging their irrelevant information by analyzing user logs on the Web server. Most of the redundant user sessions are also removed by the reduction process. For test reuse and test concurrency, it divides the user sessions obtained into different groups, each of which is called a test suite, and then prioritizes the test suites and the test cases of each test suite. So, the initial test suites and test cases, and their initial executing sequences are achieved. However, the test scheme generated by the elementary prioritization is not much approximate to the best one. Therefore, genetic algorithm is employed to optimize the results of grouping and prioritization. Meanwhile, an approach to generating new test cases is presented using crossover. The new test cases can detect faults caused by the use of possible conflicting data shared by different users.

Keywords: *User Session, Genetic Algorithm, Test Case, Test Suite, Reduction, Prioritization*

1. Introduction

Incapable Web applications can have far-ranging consequences on businesses, economies, scientific progress, health, and so on. Therefore, all the entities of a Web application, in essence, must be tested adequately to ensure that the application meets its original design specifications.

In Web application testing, some test methods and techniques were presented [1-4]. Kung, *et al*. [2] depicted an object-oriented Web Test Model to support Web application testing. Hieatt, *et al*. [1] introduced a method of acceptance testing, and developed a testing tool to show the system operations and the expected output results in XML. The approaches proposed by Kung and Hieatt, *et al*., however, focus primarily on unit testing without concerning the whole testing for Web applications. Liu, *et al*. [4] presented a data flow-based approach to testing Web applications. Most of these methods of testing Web applications are achieved through extending the testing methods for traditional software. Additionally, none of these methods [1-4] yields test data according to user sessions.

Elbaum, *et al*. [5] demonstrated the fault detection capabilities and cost-effectiveness of user session-based testing. Increment concept analysis [6] is used to analyze user sessions dynamically and minimize continuously the number of user sessions maintained. Khor, *et al*. [7] combined the genetic algorithm with formal concept analysis to trace the relationship between test data and corresponding test run.

Sthamer [8] analyzed deeply the test case optimization efficiency of different coding schemes and fitness functions of genetic algorithm for different structure-based software in his doctoral dissertation. Some researchers also studied on test case generation techniques using genetic algorithm [9-10]. However, they aimed for simplex optimization focusing on one-off optimization computation, while the testing is continuous and iterative. So, dynamically continuous optimization computation is more propitious to improve test performance. Jia, *et al*. [11] discussed the key problems of producing test data of covering designated paths using genetic algorithm, and introduced deeply the factors of influencing the genetic algorithm's efficiency through experimental results. Berndt and Watkins [12] summarized the advancement in generating data for generic algorithm-based testing recently. These studies [5,7-12] used genetic algorithm to

*This research has been partly supported by the National High-Tech Research and Development Plan of China under Grant No. 2007 AA01Z144; the Science and Technology Plan Project of the Education Department of Jiangxi Province of China under Grant Nos. GJJ10120 and GJJ10117; Shanghai Leading Academic Discipline Project of China with Project No. J50103; and the School Foundation of Jiangxi University of Finance & Economics of China under Grant No. 04722015.

Copyright © 2010 SciRes.

analyze test problems with the exception of [5], but not focusing on Web application testing. In [5], the authors discussed user session-based Web application testing, but not concerning deeply the optimization of test cases.

Different from them, this paper investigates a key problem in Web application testing: test case generation and optimization. It delves into an approach to testing and optimizing Web applications based on user sessions using genetic algorithm. When converting a user session to a corresponding test case, we preserve the user input data.

2. Collecting and Reducing User Sessions

In the logs on each Web server, each access record corresponds to a request by a user each time. The contents of the record include request source (user IP address), request time, request mode (such as GET, POST), the URL of requested information, data transport protocol (such as HTTP), status code, the number of bytes transferred and the type of client, etc. It needs to scan the logs only once to resolve the original active historic records from current access logs. However, it is difficult to organize these original records directly, which must be preprocessed. We first purge irrelevant data including the records whose status codes are erroneous (the code 200 for success, 400 for error), embedded resources such as script files and multimedia files whose extension names are .gif, .jpeg or .css, etc., to obtain the set of user sessions for primitive analysis. Then, we create user sessions through scanning the logs on Web servers. Once a new IP address occurs, a new user session is created. The sequential requests sent from this IP address are appended to the new session under the condition that the time interval of two continuous requests is not greater than max-session-idle-time pre-determined, or else another new user session begins. The set of all the user sessions is finally achieved.

There are often large volumes of user sessions collected and a user session is converted to a test case transmitted to Web server. Therefore, we can eliminate *redundant* user sessions using reduction techniques, and then preserve *necessary* user sessions. For the convenience of discussion, some important concepts are given first.

Definition 1 (URL trace). *URL trace* is the URL sequence requested by a user session.

Let α be a URL trace. Its length is the number of URLs requested in the trace, denoted by $|\alpha|$.

Definition 2 (prefix). A trace α is the *prefix* of another trace β, if and only if α is the subsequence of β and they have the same initial symbol.

Definition 3 (common prefix). If a trace is the prefix of several traces, then this trace becomes their *common prefix*.

Definition 4 (greatest common prefix). The longest common prefix in all the common prefixes of two traces is their *greatest common prefix*.

The longer the greatest common prefix of two traces is, the more similar the two traces are, *i.e.*, their similarity is higher. Let we have four traces that are γ_1 = abcdefg, γ_2 = abcdeh, γ_3 = abcd and γ_4 = cde. Then, γ_3 is the common prefix of γ_1 and γ_2, but not the greatest one. The greatest common prefix of γ_1 and γ_2 is abcde. Although γ_4 is the subsequence of γ_1 and γ_2, it is not their prefix, for the initial symbol of γ_4 is not the same as that of γ_1 and γ_2. We define a function isPrefix(α, β), which decides whether or not α is the prefix of β, *i.e.*, whether or not the URL trace requested by a user session is *redundant* corresponding to that requested by another user session. If there is a prefix, then the function returns a BOOLEAN value TRUE, or else returns FALSE. The URL trace-based user session algorithm ReduceUSession is shown in **Figure 1**. Note that, an HTTP request here is regarded as a symbol of a trace.

The ReduceUSession algorithm decides whether or not a URL trace α requested by a user session is the prefix of β requested by another user session. If it is true, then the user session corresponding to α is removed. The number of user sessions obtained using this algorithm will be reduced greatly.

The ReduceUSession algorithm is different from other

```
Algorithm: ReduceUSession
input:
    The set of user sessions Λ = {s1, …, sk}, where k is the number
    of user sessions;
    The URL trace U1, …, Uk, which are requested by s1, …, sk
    respectively;
output:
    The reduced set of user sessions denoted by Γ;
begin
    Γ = Φ;
    while (another user session that is not marked in Λ exists)
        tag1 = FALSE;
        tag2 = FALSE;
        Select a user session si that is not marked in Λ, and then
        mark it with "USED";
        for (the URL trace Uj requested by each user session sj in
        Γ)
            if isPrefix(Uj, Ui)      //the URLs requested by si is
                                     //more, so sj is redundant
                Γ = Γ-{sj};
                tag1 = TRUE;
            endif;
            if isPrefix(Ui, Uj)      //here, Γ keeps unchanged,
                                     //and si is redundant
                tag2 = TRUE;
                break;  //exit for cycle
            endif;
        endfor;
        if tag1 || (!tag1 && !tag2)     //si is necessary
            Γ = Γ ∪ {si};
        endif;
    endwhile;
    Output the reduced set of user sessions Γ;
end.
```

Figure 1. The URL trace-based user session reduction algorithm ReduceUSession

user session reduction algorithms. Most other reduction algorithms analyze each test case in the test suites one by one and remove those test cases that cannot change or affect test requirements, *i.e.*, they distinguish *redundant* and *necessary* test cases, as is too difficult to manage in practice, for it is very intractable to discriminate that the test requirements satisfied by some test cases (*i.e.*, redundant ones) are also satisfied by other test cases before the test execution using the existential algorithms. While our ReduceUSession algorithm decides whether or not a URL trace requested by a user session is the prefix of another URL trace requested by another user session. Based on this, we can identify the *redundant* test cases, as can be easily done in practice. Besides, it covers all the URLs requested by the original set of user sessions and keeps the sequence of URL requests, *i.e.*, it guarantees that the original test requirements are satisfied.

3. Grouping and Prioritizing User Sessions

The user sessions reduced by the ReduceUSession algorithm are divided into subgroups, each of which is regarded as a test suite. The goal of grouping is to reuse test cases and the testing can be executed at different platforms in parallel (or concurrently), to lessen test time and improve test efficiency. Moreover, the interacting test can be conducted between a pair of user sessions in each group (see Subsection 4.4 for the details). We try our best to keep the property for the user sessions in the same group that the URL traces requested bear greatest common prefix of a certain length. Several discontinuous integral threshold values, denoted by $\zeta_1, \zeta_2, \ldots \zeta_k$ ($\zeta_i \geq 1$, $1 \leq i \leq k$), are defined. The user sessions, the lengths of whose greatest common prefix of URL traces requested fall in between a certain threshold values, are grouped together. Fox example, let we have three threshold values $\zeta_1 = 2$, $\zeta_2 = 4$ and $\zeta_3 = 7$, then the user sessions are divided into four groups that are S_1, S_2, S_3 and S_4, the lengths of whose greatest common prefix (denoted by α) are $|\alpha| \leq 2$, $2 < |\alpha| \leq 4$, $4 < |\alpha| \leq 7$ and $|\alpha| > 7$ respectively. The test cases in these four groups can be executed at different platforms in parallel (or concurrently), and the greatest common prefix is also reused in each group. Notice that this is not the unique grouping way. For example, in the four traces γ_1 = abcdefg, γ_2 = abcdeh, γ_3 = abcd and γ_4 = cde, we can divide them into two groups $\{\gamma_1, \gamma_2\}$ and $\{\gamma_3, \gamma_4\}$, the lengths of whose greatest common prefix are $4 < |\alpha| \leq 7$ and $|\alpha| \leq 2$ respectively, or another two groups $\{\gamma_1, \gamma_2, \gamma_3\}$ and $\{\gamma_4\}$, the lengths of whose greatest common prefix are $2 < |\alpha| \leq 4$ and $|\alpha| \leq 2$ respectively. Of course, it is unnecessary to require that the greatest common prefix of URL traces requested by any two user sessions in each group fall in between a certain threshold values; it is recommended that most of them satisfy this property (a percentage can be pre-designed or generated randomly for the measurement). A compromise way needs to be found to classify these URL traces (or test suites), *i.e.*, a tradeoff should be found between grouping and concurrent testing and test reuse. Suppose that N user sessions are divided into K groups. In general, there is almost the same number of user sessions in one group as that in another, *i.e.*, the number equals approximately to $\left\lceil \frac{N}{K} \right\rceil$. Additionally, this way of grouping is preparatory, called *elementary grouping*.

The common prefix indicates the users' common events, or the same or similar operations. It also shows that the users bear the same or similar interests. The longer the common prefix is, the more evident it is, as is the case of most users. In addition, there is a special group of user sessions, the length of whose greatest common prefix of URL traces requested is shortest. This group of user sessions often indicates different URL requests, which represent distinct requirements for a Web application. In these sessions, many aberrant events often occur with unwonted input data. They belong to boundary cases, which are very easy to go wrong for the Web application. Herein, we prioritize test suites. The test suite with shortest length of common prefix ranks first, then all the other test suites are arranged according to their lengths of common prefix in descending order. So, the test suite in final position is that whose length of common prefix is last but one. In the test suites S_1, S_2, S_3 and S_4, the lengths of whose common prefixes are $|\alpha| \leq 2$, $2 < |\alpha| \leq 4$, $4 < |\alpha| \leq 7$ and $|\alpha| > 7$ respectively, if we prioritize them, then the test executing sequence for those test suites are S_1, S_4, S_3, S_2. That is to say, the test cases in S_1 are executed first, then the test cases in S_4 and S_3 are executed respectively and finally, the test cases in S_2 are executed. In each test suite, the test cases are prioritized according to the coverage ratios of URLs requested, *i.e.*, the test case with longer URL trace requested is executed earlier. If the lengths of URL traces of several test cases in the same test suite are equal, then they are randomly executed.

Our approach of prioritization is different from others. The existing methods of prioritization add the strongest test case, which is of the maximal use for the coverage ratio of test requirements, to the new test suite. Those methods aim to execute earlier the test cases of high priority than those of lower priority, to satisfy some test requirements as soon as possible. These methods, however, are often difficult to find the (nearly) strongest test case each time before test run, while our approach divides test cases into several groups according to the idea of common prefix before prioritizing them. It is more convenient for the testers to selectively execute the test cases of some group by grouping all the test cases first, to detect some

Copyright © 2010 SciRes

types of faults, for the same or similar types of errors are often detected by those test cases in the same groups. In addition, test cases can be reused by grouping and the testing can be executed at different platforms in parallel (or concurrently), to lessen test time and improve test efficiency. Moreover, we have considered a special type of user sessions, the length of whose greatest common prefix of URL traces is shortest. This group of user sessions often contains specific requests, which are the primary source for errors.

4. Testing Web Applications Using Genetic Algorithm

After the process of grouping and prioritization above, we obtain several initial test suites and test cases with the initial executing sequences. However, the test scheme generated by the elementary prioritization is not fast in finding faults and can not satisfy the requirements earlier, *i.e.*, it is not much approximate to the best one. Therefore, genetic algorithm is employed further to optimize the grouping and prioritization.

In 1975, an American professor Holland first proposed the idea of genetic algorithm systematically [13]. It has attracted a large number of researchers and extended into those aspects of optimization, search and machine learning with a solid theoretic foundation. The genetic algorithm focuses on all the individuals in one population, and uses random techniques to search efficiently for a coded parameter space. Selection, crossover and mutation are the basic operators in genetic algorithm; parameter coding, the setting of initial population, the design of fitness function, the selection of genetic operations and control parameters consist of the critical part of genetic algorithm. As a global optimization search algorithm of high efficiency, the genetic algorithm has distinctive advantage in solving the difficult problems in the domains of big space, multiple peak, nonlinearity and parallel processing, etc.

In the following, we test Web applications using genetic algorithm to further optimize the initial test suites and test cases, and their initial executing sequences, in order to achieve better test suites and test cases that satisfy test requirements. The process of yielding a population of next generation using three basic operators that are selection, crossover and mutation once is called *an iteration*. To obtain a good result, much iteration is repeated. The following introduces the process of selection, crossover and mutation.

4.1 Selection

A pair of individuals is selected from a parent population with the probability of p_s. The probability that an individual is selected is in direct proportion to its fitness value, as is often implemented using the strategy of roulette wheel [13]. In selecting, the individual of high fitness value is duplicated into the population of next generation directly. The higher the fitness value of an individual is, the higher the probability of yielding its offspring is, as shows that it is more appropriate to the expected result. Let we get K test suites (constituting the initial population), which are S_1, S_2, …, S_K of the descending order according to the prioritization technique. In practice, we combine the error coverage ratios of test suites and the cost of test run to design fitness function. The fitness value is listed as f_1, f_2, …, f_K from high to low, where f_i is the fitness of S_i ($1 \le i \le K$). The probability that an individual is selected equals to the resulting value that its fitness value divides the sum of fitness values of all the individuals, *i.e.*, the selected probability of S_i, denoted by p_s^{Si}, is $f_i \Big/ \sum_{j=1}^{K} f_j$. Obviously, the sum of the selected probabilities of all the K test suites equals to 1.

According to the discussion above, the fitness f_1 and f_2 corresponds to two special test suites S_1 and S_2, the lengths of whose greatest common prefixes are shortest and longest respectively. S_1 and S_2 are selected to be the individuals of next generation directly (in practical use, we can also select more than two individuals of high fitness to become the next individuals); in case they do not be selected. Now, we randomly yield two numbers that are $g_1, g_2 \in [0, 1]$, and randomly select two test suites that are S_i and S_j whose probabilities are not less than g_1 and g_2 respectively. The two new test suites S_i' and S_j' (the individuals of next generation) are generated through crossovering and mutating their parents S_i and S_j (see Subsections 4.2 and 4.3). Repeat the process of selection, until adequate test suites of next generation are yielded. One point should be emphasized that some test suites of higher probabilities may not be selected as parents, while others of lower probabilities are selected. This case is reasonable and accords with the theory of biological evolution in nature, because any thing has its necessity and occasionality at the same time.

4.2 Crossover

The genes chain of two parent individuals selected are crossovered with the probability of p_c using the TSCrossover algorithm, where p_c is a system control parameter. The new individuals after crossovering are used to replace their parents. The TSCrossover algorithm is shown in **Figure 2**.

Let we have two test suites $S_i = < c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9 >$, which contains 9 test cases and $S_j = < c_{10}, c_{11}, c_{12}, c_{13}, c_{14}, c_{15}, c_{16} >$, which contains 7 test cases. If the position of crossovering is 3, then two new individuals of next generation after crossovering S_i and S_j are: $S_i' = < c_1, c_2, \mathbf{c_{12}, c_{13}, c_{14}, c_{15}, c_{16}}, c_8, c_9 >$ and $S_j' = < c_{10}, c_{11}, \mathbf{c_3, c_4, c_5, c_6, c_7} >$, where the test cases indicated in bold face are those interchanged one by one from corresponding test cases in parent test suites.

4.3 Mutation

Each bits of the genes chain of new individuals are mutated with the probability of p_m using the TSMutation algorithm, where p_m is a system control parameter. The probability of mutation is often small, or else it will raise questions over the system stability. The populations can be prevented from stagnating through mutation. If there is no mutation, then the test data of new populations will be confined to the initial values. The mutation process is conducted respectively using the TSMutation algorithm for the test suites S_i' and S_j' crossovered using the TSCrossover algorithm. The new individuals after mutating are employed to replace those before mutating. The TSMutation algorithm is shown in **Figure 3**.

Let $p_m = 0.015$ and S_i' = <c_1, c_2, c_{12}, c_{13}, c_{14}, c_{15}, c_{16}, c_8, c_9>, which is one of the test suites after using the TSCrossover algorithm. We randomly generate some data: $g(c_1) = 0.246$, $g(c_2) = 0.580$, $g(c_{12}) = 0.025$, $g(c_{13}) = 0.012$, $g(c_{14}) = 0.632$, $g(c_{15}) = 0.073$, $g(c_{16}) = 0.431$, $g(c_8) = 0.193$ and $g(c_9) = 0.059$. For p_m is not less than $g(c_{13})$, c_{13} is mutated, *i.e.*, the positions of c_{13} and c_{12} are interchanged. The test suites S_i' after mutating becomes < c_1, c_2, $\mathbf{c_{13}}$, $\mathbf{c_{12}}$, c_{14}, c_{15}, c_{16}, c_8, c_9 >.

4.4 The Interacting Testing among User Sessions

Some new test cases, which contain the requested information of different users, can also be generated using crossover. The goal of the new test cases is to detect errors caused by the use of possible conflicting data shared by different users. The crossover way shows the idea of information interchange among different user sessions. Let S be a test suite, the steps of generating a test case using crossover are:

1) for a test case c in S, randomly generate a number $g \in [0, 1]$, if a given crossover probability is not less than g, then

a) copy the requests from r_1 to r_i in c to an interim test case, where i is a random number, $i \in [1, |c|]$; $|c|$ is the length of URL trace in c;

b) select a test case d (different from c) in S randomly, and then search r_j in d reversely, which is the first one to have the same URL as r_i. If not found, then another test case (also denoted as d) is selected, until it is found or up to a given time;

c) if d is found, then all the requests after r_j in d are appended to the interim test case, which is the new test case generated; otherwise go 2);

2) repeat 1) until each test case in the test suite is processed.

We search the requests in test case d with an inverse search method, for the length of greatest common prefix of two test cases c and d in the same test suite is often greater than 1; when the corresponding URL trace from r_1 to r_i is the prefix of this greatest common prefix, we can not obtain different test cases if using the sequential search method. For example, given two URL traces $c = u_1u_2u_4u_3u_5u_6u_8u_7$ and $d = u_1u_2u_4u_3u_7u_4u_5u_9$, we randomly copy $u_1u_2u_4$ from c to an interim test case t firstly (*i.e.*, t equals to $u_1u_2u_4$ temporarily). Here, $I = 3$ and $r_i = u_4$. Now, we search r_j in d reversely, which is the first one to have the same URL as r_i and then we have $j = 6$ and $r_j = u_4$. So, all the requests after r_6 in d (*i.e.*, u_5u_9) are appended to t such that $t = u_1u_2u_4u_5u_9$. If searching the requests in d with a sequential search method, we have $j = 3$ and $r_j = u_4$. This time, if copying all the requests after r_3 in d (*i.e.*, $u_3u_7u_4u_5u_9$) to t, we have $t = u_1u_2u_4u_3u_7u_4u_5u_9$, which equals to d itself. And no new test case is generated. The

```
Algorithm: TSCrossover
input:
    Parent test suites Si and Sj;
    The crossover probability of pc;
output:
    Si' and Sj' of next generation;
begin
    Si', Sj'←Si, Sj;
    Randomly generate a number g∈[0, 1];
    if pc is not less than g
        Randomly generate an integral number g'∈(0, min(|Si'|,
        |Sj'|)];    //|Si'| and |Sj'| are the number

                    //of test cases in Si' and Sj' respectively
        Interchange the test cases in Si' and Sj' from the position of
        g' one by one,
        until no more test case needs to be interchanged in any of
        them;
    endif;
    Output Si' and Sj';
end.
```

Figure 2. The crossover algorithm TSCrossover for test suites

```
Algorithm: TSMutation
input:
    The test suites Si' and Sj' after using TSCrossover algorithm;
    The mutation probability of pm;
output;
    The test suites Si' and Sj' after mutating;
begin
    for each S in {Si', Sj'}
        for each c in S
            Randomly generate a number g∈[0, 1];
            if pm is not less than g
                Interchange the positions of c and the test case
                before c;
            endif;
        endfor
        Output S;
    endfor
end.
```

Figure 3. The mutation algorithm TSMutation for test suites

Copyright © 2010 SciRes

reason is that the corresponding URL trace (*i.e.*, $u_1u_2u_4$) from r_1 to r_i (i = 3) in c is the prefix of greatest common prefix (*i.e.*, $u_1u_2u_4u_3$) of c and d.

In addition, it is often not to mutate a test case, for the new generated test case after exchanging the two adjacent requests r_i and r_{i+1} makes no sense. The possible reason is that the former request r_{i-1} may never reach the request r_{i+1} (note that, this time r_i becomes the next request of r_{i+1}).

5. Experimental Analysis

Consider a typical miniature Web application developed to demonstrate our approach: the SWLS (Simple Web Login System) was shown in **Figure 4**.

Starting at the first page (indicated by a dashed arrow, reasonably, a *blank* page can be used to request for the first page of a Web application), *i.e.*, a *home page* (p_1), the user can enter into the *news page* (p_2) to list the news by clicking on the *view* link, or enter into the *login page* (p_3) by pressing the *login* button. In page p_3, the user enters the *userid* and *password,* and presses the *submit* button. Upon this pressing, the *userid* and *password* are sent to the Web server for authentication. A *logged page* (p_4) will be loaded if both *userid* and *password* are correct. On the contrary, an *error page* (p_7) containing an error message is displayed if at least one of the submitted values for *userid* and *password* is wrong. From the *logged page*, it is possible to go to *info page* (p_5) for secure information viewing by just clicking on the *browse* link. The user can click on the intra-page link *continue* to view the different parts of the same page p_5 if it is too long. A *logout page* (p_6) will be displayed when the user presses the *exit* or *logout* button. Then, the user may come back to the *home page* for login again. Note that each time the *login page* is displayed; both the *userid* and *password* fields should be initialized to be empty.

We inject only one fault in each page respectively, so at least 7 faults exist totally (there may be other faults in the Web application originally). A set of user sessions are obtained after scanning user logs on the Web server and purging their irrelevant information; then 89 meaningful user sessions are finally created after scanning them again. Only 17 user sessions are used to generate test cases after the reduction of the meaningful user sessions using the URL trace-based ReduceUSession algorithm; and the reduction ratio reaches 80.9%. Impersonally, more user sessions there are for a given Web application, much higher is the reduction efficiency, for more user sessions mean higher possibility that the URL trace requested by a user session is the prefix of that requested by another in the view of statistics. The set of 17 user sessions[†] (or test cases) is denoted by Γ1. After grouping and prioritizing Γ1, an initial executing sequence <S_1, S_2, S_3> of test suites is obtained, which is denoted by Γ2, where S_1, S_2 and S_3 contain initial executing sequences of 7, 6 and 4 test cases respectively. We achieve the final executing sequence <S_1', S_2', S_3'> of test suites using genetic algorithm for further processing, which is denoted by Γ3, *i.e.*, S_1', S_2' and S_3' is obtained through crossovering and mutating S_1, S_2 and S_3.

Now, we run the test suites (and test cases) in Γ1, Γ2 and Γ3 for the Web application respectively, and find all the 7 faults injected as well as an additional fault (*i.e.*, the user may browse p2 just by directly entering its address in the URL address bar). The executing time of Γ2 and Γ3, however, is much shorter than that of Γ1, and it takes less time of Γ3 than that of Γ2 too. This means that the test suites (and test cases) grouped and prioritized run faster than those, which are not grouped and prioritized; and that the test suites (and test cases) processed further by genetic algorithm are much faster. It is predictive that the test approach proposed in this paper will yield more evident positive effects for larger Web applications.

6. Concluding Remarks and Future Work

Generating test cases of high quality is the premise of Web application testing. The approach in this paper captures a series of user events, *i.e.*, the sequences of URLs and name-value pairs in Web server (s). It then employs the reduction, grouping, prioritization and genetic algorithm to yield test cases and optimizes them. Compared to the methods of capturing user events in clients, our approach is very effective when a large volume of users exist, and it is a Web application testing method of high efficiency. The main contributions include:

1) an approach to generating and optimizing test cases is proposed for Web application testing based on user sessions using genetic algorithm.

2) several important definitions such as URL trace, prefix, common prefix, greatest common prefix, are given. These definitions are convenient for reducing and grouping user sessions.

3) a URL trace-based reduction algorithm is designed. The user sessions acquired are lessened greatly using the

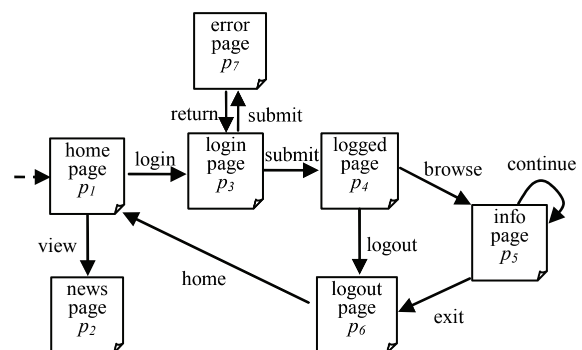

Figure 4. A simple web login system

[†] For the convenience of test run, we have preprocessed the user sessions appropriately.

algorithm. However, it covers all the URLs requested by the original set of user sessions and keeps the sequence of URL requests.

4) an approach to grouping and prioritizing user sessions is presented, as can improve the efficiency of test run.

5) an approach to generating new test cases is proposed using crossover. The new test cases generated include information requested by different users and can detect errors caused by the use of possible conflicting data shared by different users. That's to say, the crossover indicates the idea of information interchange among different user sessions, which helps to test the interacting of user sessions.

6) a strategy of test reuse and concurrence is provided, as can decrease the time of test run greatly as well as lessen test cost.

Web application testing is much complex systems engineering. It is not easy to acquire an effective and practical test scheme. Our approach only evaluates a test case according to its test coverage ratio. However, many factors need to be considered, such as the running cost of each test case itself (for example, CPU time), including the actual running time, loading time, time to save test state, and the influence of different test criteria on a test case. All these questions need to be answered in future research.

REFERENCES

[1] E. Hieatt and R. Mee, "Going Faster: Testing the Web Application," *IEEE Software*, Vol. 19, No. 2, 2002, pp. 60-65.

[2] D. C. Kung, C. H. Liu and P. Hsia, "An Object-Oriented Web Test Model for Testing Web Applications," *Proceedings of the* 1*st Asia-Pacific Conference on Web Applications*, New York, 2000, pp. 111-120.

[3] J. Offutt, Y. Wu. and X. Du, *et al*., "Bypass Testing of Web Applications," *Proceedings of the* 15*th IEEE International Symposium on Software Reliability Engineering*, Bretagne, November 2004.

[4] C. H. Liu, D. C. Kung and P. Hsia, "Object-Based Data Flow Testing of Web Applications," *Proceedings of the* 1*st Asia-Pacific Conference on Quality Software*, Hong Kong, 2000, pp. 7-16.

[5] C. Elbaum, G. Rothermel and S. Karre, *et al*., "Leveraging User Session Data to Support Web Application Testing," *IEEE Transaction on Software Engineering*, California, May 2005.

[6] R. Godin, R. Missaoui and H. Alaoui, "Incremental Concept Formation Algorithms Based on Galois (Concept) Lattices," *Computational Intelligence*, Vol. 11, No. 2, 1995, pp. 246-267.

[7] S. Khor and P. Grogono, "Using a Genetic Algorithm and Formal Concept Analysis to Generate Branch Coverage Test Data Automatically," *Proceedings of the International Conference on Automated Software Engineering*, Austria, 2004, pp. 346-349.

[8] H. H. Sthamer, "The Automatic Generation of Software Test Data Using Genetic Algorithms," PhD. Dissertation, University of Glamorgan, Wales, 1996.

[9] D. Berndt, J. Fisher and L. Johnson, *et al*., "Breeding Software Test Cases with Genetic Algorithms," *Proceedings of the* 36*th Hawaii International Conference on System Sciences*, 2003, pp. 17-24.

[10] R. P. Pargas and M. J. Harrold, "Test-Data Generation Using Genetic Algorithms," *Journal of Software Testing, Verification and Reliability*, Vol. 9, No. 4, 1999, pp. 263-282.

[11] X. X. Jia, J. Wu, M. Z. Jin, *et al*., "Some Experiment Analysis of Using Generic Algorithm in Automatic Test Data Generation," in Chinese, *Journal of Chinese Computer Systems*, Vol. 28, No. 3, 2007, pp. 520-525.

[12] D. J. Berndt and A. Watkins, "Investigating the Performance of Genetic Algorithm-Based Software Test Case generation," *Proceedings of the International Symposium on High Assurance Systems Engineering*, Florida, 2004, pp. 261-262.

[13] J. H. Holland, "Adaptation in Natural and Artificial System," University of Michigan Press, Michigan, 1975.

Copyright © 2010 SciRes