Leveraging task-based data to support functional testing of web applications

Flávio Rezende de Jesus
POSCOMP
Univ. Federal de Itajubá
flavio.rezende@gmail.com

Leandro Guarino de Vasconcelos
Instituto Nacional de Pesquisas Espaciais
leandro.guarino@lit.inpe.br

Laércio A. Baldochi Jr.
Instituto de Matemática e Computação
Univ. Federal de Itajubá
baldochi@unifei.edu.br

ABSTRACT

Testing is paramount in order to assure the quality of a software product. Over the last years, several techniques have been proposed to leverage the testing phase as a simple and efficient step during software development. However, the features of the web environment make application testing fairly complex. The existing approaches for web application testing are usually driven to specific scenarios or application types, and few solutions are targeted for testing the functional requirements of applications. In order to tackle this problem, we propose a task-based testing approach that provides high coverage of functional requirements. Our technique consists of reassembling classical graph algorithms in order to generate all the possible paths for the execution of a task. Performed experiments indicate that our approach is effective for supporting the functional testing of web applications.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: [Testing and debugging – testing tools]

General Terms

Algorithms, Experimentation

Keywords

Web application testing, tasks, graph algorithms

1. INTRODUCTION

In recent years the web has become the universal medium for the development of software applications. Besides universal access from any machine connected to the Internet, web applications have no installation costs, may be upgraded automatically and are independent of client's operating systems. On the other hand, the use of server and browser technologies that are constantly evolving make web applications error-prone. In fact, web-based systems are infamous for being poorly developed [12]. As a result, a large number of applications present defects, causing both financial and credibility losses to the software industry.

Software testing is the most effective way in order to prevent defects. However, the features of the web environment make application testing fairly complex, as (i) web applications are distributed in a client server architecture and relies on request/response calls to synchronize their state; (ii) they are heterogeneous, i. e., usually built with different languages, both in client and server sides; and (iii) they have a dynamic nature, possessing non-deterministic characteristics in some scenarios [5].

Over the last decade different techniques have been proposed in order to cope with the complexity of web application testing. Many of the existing techniques fall in the category of model-based testing, where a model of the web application is built and test cases are derived from this model. There are also techniques that exploit logged user sessions to generate test cases. Finally, there are crawling-based approaches that perform the systematic exploration of web applications, allowing the generation of navigational maps that can be used to produce test cases.

Whichever the used technique, the works reported on the literature present limited test coverage. Therefore, there is no "one size fits all" solution. Each approach is more adequate for a certain scenario or application type. For instance, there are solutions targeted to JavaScript applications [2, 4], others targeted to AJAX applications [9, 11], and, more recently, to mobile applications [8].

A common point among the existing approaches is that they are not targeted to cover functional testing. Some works report coverage of functional testing to some extent, but, in general, no approach is guided by the functional requirements of applications. An exception is the work by Thummalapenta et al. [15], which aims at automating the creation of functional tests based on business rules. Using a two-step black-box approach, their solution first crawls the application GUI and creates an abstract state transition diagram, which is then used in the second step to identify rule relevant abstract paths. Although more effective than existing solutions, this approach is not able to cover all functional requirements of an application. Moreover, as this approach is based on crawling the application's GUI, it is rather computationally costly, specially for large web applications with unbounded number of GUI states.

In this paper we propose a technique for functional testing that is able to exercise the functional requirements of a web application without using costly crawling approaches.

SAC'15 April 13-17, 2015, Salamanca, Spain.

http://dx.doi.org/10.1145/2695664.2695917

Our solution leverages techniques that exploits user-session logs [6, 14]. However, instead of gathering real user data, we prefer to record tasks performed by the application developer using a tool called UsaTasker [17]. This tool allows developers to define tasks by simply interacting with the application's GUI.

UsaTasker provides facilities for the visualization and management of captured tasks, allowing (i) the definition of sequence of events in which each event may occur in any order; (ii) the definition of sequence of events that may be repeated several times; and (iii) the definition of optional events. In this way, it is possible to specify, for instance, that the fields of a form may be filled in any sequence (i), that certain steps of tasks, such as putting products on a shopping cart may be repeated several times (ii) and, finally, that a certain field of a form is optional and, therefore, may be left blank (iii).

Our approach for generating test cases consists in expanding the recorded navigational path by exploiting the features of our navigational model. The resulting navigational graph contains all paths that allows the correct execution of the recorded task. Therefore, by replaying all these paths it is possible to test the functionality expressed by the recorded task. Repeating this procedure for all tasks in an application assures very high coverage of functional testing.

In this paper we detail the procedure for generating expanded navigational graphs for recorded tasks and show how these graphs are used to derive test cases. We also present an experiment that demonstrates that our approach is effective for conducting positive tests in web applications.

This paper is organized as follows. Section 2 presents UsaTasker and gives an overview of previous work developed by the authors. Section 3 details our approach for deriving test cases from recorded tasks. Section 4 presents an experiment performed to evaluate our approach. Section 5 presents related work and discusses the advantages of our solution when compared to the existing approaches. Finally, Section 6 presents our final remarks.

2. PREVIOUS WORK

The work presented in this paper builds upon previous research on usability evaluation. As a result of this research, we developed USABILICS, a task-based system that performs the automatic remote evaluation of usability in web applications [16]. USABILICS evaluates the execution of tasks by calculating the similarity among the sequence of events produced by users and those previously captured by evaluators. This evaluation provides a metric for the efficiency and for the effectiveness of each evaluated task. We named this metric the *usability index* of a task. USABILICS also provides recommendations detailing actions to be performed in order to solve detected usability problems.

One of the goals of USABILICS was to perform usability evaluation with a minimum burden on the application developer (which is the application evaluator in most cases). To achieve this goal, we implemented UsaTasker [17], a task definition tool that supports the management of tasks targeted to be evaluated in a web application. UsaTasker allows evaluators to define tasks by simply interacting with the application's GUI. After recording a task, our tool provides facilities for the management of the captured events, allowing (i) the definition of sequence of events in which each event may occur in any order; (ii) the definition of sequence of events that may be repeated several times; and (iii) the definition of optional events.

UsaTasker provides a GUI which presents a captured task as a sequence of boxes, where each box represents an event of the task. In order to define an event as optional, all the evaluator needs to do is to select the desired box using the mouse and change its property from *mandatory* to *optional*. Changing the ordering of events is also very simple. To perform this action, the evaluator selects the events (consecutive boxes) in which the precedence relation is not wanted. When an event is marked as without precedence, it is displayed with a yellow background in UsaTasker's GUI. Finally, to allow the repetition of events within a task, UsaTasker presents a feature that allows selecting sequences of events that may be repeated.

After recording and editing a task, a directed graph is generated containing nodes that represents events and edges that express the sequence in which each event occurs. Besides being used in usability evaluation, we envision that this directed graph can be used to support the functional testing of web applications. Therefore, we designed and implemented an extension to UsaTasker, which we called UsaTasker++.

3. USATASKER++

We advocate in this paper that, by exploiting the features of our navigational model, it is possible to derive test cases from the directed graph that represents a tasks. Taking into consideration that a given task may have optional, out of order and cyclic events, it is possible to execute the task in a myriad of different ways. In other words, there are several different paths from the starting point (first event) to the completion of the task (last event). The goal of UsaTasker++ is to discover all these paths, making each existing path a test case.

In order to produce all the possible paths with common starting and ending events, we developed a set of algorithms that process a directed graph taking into consideration the features of our navigational model. The procedure to process the graph use different techniques, reassembled in the following steps:

1. graph calibration: creates an adjacency list that represents the graph;
2. path discovery: finds all basic paths in the graph;
3. reduction method: removes invalid paths;
4. out of order processing: creates additional paths to represent the out-of-order events;
5. cycle processing: creates additional paths to add the cycles iterations.

3.1 Graph calibration

In this step, the original graph from UsaTasker is converted to an adjacency list. Then, the adjacency list is refactored in multiple iterations to effectively address the three types of events: optional, cyclic and out-of-order.

An optional event generates an extra edge between the event before and another after the optional vertex. In this case, the user can skip an event, going directly from the previous action to the one after it, as shown in Figure 1 (A).

An example of this characteristic is a web application in which the user can optionally fill out the middle name field or mark a subscribe check-box. A restriction to this type of event is that the initial event and the last event of a task must be mandatory (non optional) since they define the boundaries of the graph. These boundaries are used to determine when to start and when to stop processing.

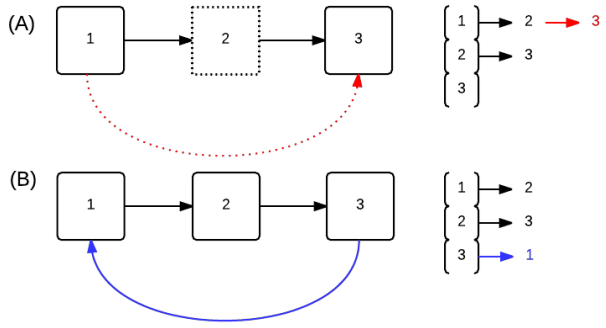

Figure 1: Graph and Adjacency List: (A) extra edge due to optional event 2; (B) extra edge due to cyclic event 3.

The cyclic event connects one vertex to another located in a previous position in the graph. This is used in case of a repeated set of events in the same flow. Figure 1 (B) shows an example of this type. Cycles are only processed if the number of elements between the vertexes is greater than one. This is just to differentiate from the out of order events, which have cycles between two vertexes next to each other. There is no restriction to the destination vertex of a cycle event in being optional or out of order. As an example, cycle events in an e-commerce application can be the addition of multiple products in a shopping cart.

An out-of-order event must always coexist with another, generating several extra edges. Take the example of Figure 2. The five events 1, 2, 3, 4 and 5 – in which 2, 3 and 4 are out-of-order – indicate that it is possible to have many sequences of events: 1-2-3-4-5; 1-3-2-4-5, 1-4-3-2-5, and so on. In such case, there are several additional edges compared to the original set. This overall step is divided in four phases.

In the first phase, shown in Figure 2 (A), the out-of-order elements next to each other are gathered in a list. One graph can have several out-of-order lists as long as the lists are separated by an element that is neither optional nor out-of-order. During the second phase, the vertex before the out-of-order list is connected to all the vertexes in the list – see Figure 2 (B). In the third phase, all the vertexes in the out-of-order group are connected to the vertex after the group, as shown in Figure 2 (C). The final phase of this step, exemplified in Figure 2 (D), is when each vertex of the group is connected to all the others inside the group. An example of this type of event in a web application is a set of text fields that can be filled in any order.

3.2 Path discovery

The second step detailed in this paper is to generate the paths that represent different sequences of events. With the refactored Adjacency List completed in the previous step, all paths are defined using Algorithm 1, which is a manipulated DFS (Depth-First Search) algorithm.

The manipulation done in this algorithm that makes it different from the original DFS is that it contains triggers

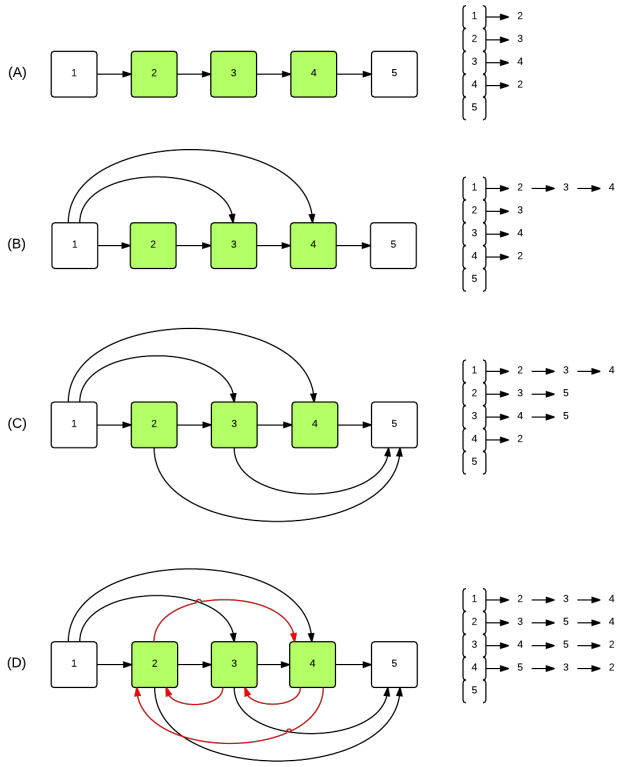

Figure 2: Out of order elements identified in green: (A) List generation; (B) Previous vertex is connected to all elements in the list; (C) Each element in the list is connected to the vertex after the list; (D) Each vertex is connected to each other in the group.

(such as "is last" in line 4), which makes it possible to output a sequence of events, not just a DFS tree.

A stack (LIFO) defined as L in the algorithm is used to represent the current sequence of events. Every time the algorithm moves forward in the graph, the event is inserted in the stack (line 2). When it moves back to a prior event, the event is removed from the stack (line 19). By doing this, the algorithm is always on hold of the sequence of events that represents the path to the current vertex.

Another difference compared to the original DFS algorithm is that this algorithm resets the edges' visited-state flag after reaching the final element of the graph (line 7). This way, the same vertex can be visited more than once in case a different path can reach it.

When the algorithm is processing each element in the path, it detects if the next element is really a further element (line 10), otherwise it considers the path as a cycle. During the graph creation, each vertex is added using a unique ID in increasing order. That way, it is guaranteed that a further element in the graph is really after a previous one. If the algorithm reaches a smaller ID, it means that it is going through a cyclic path, so the elements involved are stored on a specific cycle list for later processing (line 18).

3.3 Reduction method

At this point the adjacency list is created and the paths are depicted. Unfortunately, most paths created during the

Algorithm 1: Algorithm *ManipulatedDFS*

Input: Event E, linear list of events L, sequence of events Ps, index of the current path p, adjacency list Adj
Output: list of paths found Ps, hash-map C of vertex and adjacent

```
1  set E as visited ;
2  push E to L ;
3  add E to Ps(p) ;
4  if E is last event in path then
5      increment p ;
6      add new sequence to Ps ;
7      reset all in Adj to not visited ;
8  firstAdj ← true ;
9  foreach event adj in Adj(E) do
10     if a > E then
11         if firstAdj is false then
12             foreach event a in L do
13                 insert event a into Ps(p) ;
14             firstAdj ← false ;
15             if adj is not visited then
16                 ManipulatedDFS(adj, L, Ps, p, Adj) ;
17     else
18         add E and adj to C ;
19 remove last from L ;
```

path discovery phase are invalid. The reason is that during the out of order processing, several edges were created to represent the out of order property. Since all elements in this type of set are connected, many paths that skip the mandatory elements become available.

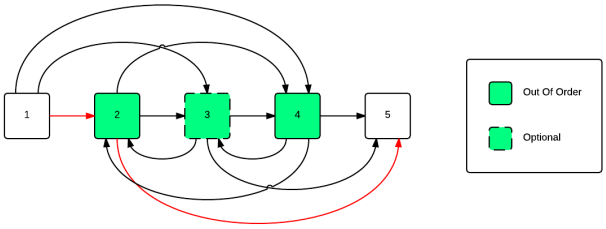

Figure 3: Invalid path

Consider the example in Figure 3, where 2, 3 and 4 are out-of-order and 3 is optional. The path 1-2-5 is an acceptable course for the DFS algorithm, but it cannot be considered as a valid test scenario, since it does not contain the non-optional event 4.

During this phase, the reduction method walks through all the paths and checks if all the non-optional elements are present. If any is missing, the path is discarded.

3.4 Out of order processing

The goal of this step is to find the variations of a single path, based on the out-of-order events. If a path contains an out-of-order set, the permutation algorithm [7] is called to generate all the possible sequences that the set can have. Then, the system replicates the path as many times as the number of variations, keeping the external elements of the set and changing the sequences.

In the previous example shown in Figure 3, the out-of-order set 2-3-4 generates the following sequences after the permutation call: [2,3,4]; [2,4,3]; [3,2,4]; [3,4,2]; [4,2,3]; [4,3,2]. The external elements are 1 and 5, so the path is replicated six times, one for each variation. The final result would be the paths derived from the out-of-order set as shown below.

```
1 - 2 - 3 - 4 - 5
1 - 2 - 4 - 3 - 5
1 - 3 - 2 - 4 - 5
1 - 3 - 4 - 2 - 5
1 - 4 - 2 - 3 - 5
1 - 4 - 3 - 2 - 5
```

Note that at this point, if any element of the out-of-order set is optional, the DFS processing detailed in the previous sections already defines a path with and another without the optional element. Therefore, permutation is executed separately for each path, considering only the elements present in the set. In the example of Figure 3, permutation is executed for the path 1-2-3-4-5, and later executed for 1-2-4-5. This last path is also processed since the vertex 3 is optional.

The permutation algorithm interchanges the position of elements to generate the next permutation. The output of this code is the list of all permutations. For example, the set 1,2 outputs [1,2] and [2,1]; the set 2,3,4 outputs [2,3,4]; [2,4,3]; [3,2,4]; [3,4,2]; [4,2,3]; [4,3,2].

Besides the permutation process, it might be required to handle more than one out-of-order set in a single path. Note that if each set is handled separately, the permutation algorithm will only generate paths varying the events within the group. To manage this scenario, we created Algorithm 2 – *ProcessOutOfOrder* – which performs the permutation between more than one out-of-order set in a single path.

Algorithm 2: Algorithm *ProcessOutOfOrder*

Input: list P of events of the path
Output: list N of new paths

```
1  permMap ← mapOOOSequences(P) ;
2  foreach path p in permMap do
3      foreach OOO sequence ooo in p do
4          permGroup ← permute(ooo) ;
5          numPerm ← numPerm× number of
           permutations in permute(ooo) ;
6      for iterator i=0 to number of permGroup do
7          for iterator index=0 to numPerm do
8              for iterator k to permGroup(i) do
9                  foreach sequence s in size of
                   permGroup(i) do
10                     newPath ← beginning of sequence +
                       sequence(k) of permGroup(i) + end
                       of sequence ;
11     add newPath to N ;
12 return N ;
```

At first, Algorithm 2 uses the Fundamental Counting Principle from elementary algebra to come up with all the possible variations in a path with the out-of-order elements (line 4). So it defines the number of variations by multiplying the amount of permutations of each out-of-order set. Take

the example of Figure 4 (A), the path 1-2-3-4-5-6-7-8-9-10-11, with the out-of-order sets 2, 3, 5, 6, 7 and 9,10, has 24 possible variations. The results can be seen in Figure 4 (B).

The next step of the algorithm is to build the permutation groups, one for each out-of-order set (line 6). The idea is to spread the permutations within the maximum variations defined by the Fundamental Counting Principle, as shown in Figure 4 (B). For example, if the group has 2 permutations and the overall maximum permutations is 24, half of the maximum will be set with the first permutation, the other twelve elements will be set with the second permutation, as shown in Figure 5(A).

While processing the second group and so on, the algorithm spreads the group permutations repeatedly until it reaches the overall maximum (line 7). Figures 5 (B) and 5 (C) illustrate these permutations. After all groups are processed, all the possible permutations are fulfilled with the correct variation, group by group.

3.5 Cycle processing

The idea of the Algorithm 3 is to add a cycle in all paths that has already been processed and that could have a cycle. To do this, the algorithm checks the presence of a cycle event in all the paths already processed (line 4). If the cycle event is present, the path is duplicated and a cycle is connected between the cycle event and its origin (lines 10 to 13).

This algorithm was designed to allow the user to select the number of cycle iterations that should be included in the test sequences (via NC input). The code starts with two *for* statements to match the cycles with the corresponding paths (line 1 and 3). If a cycle vertex is present in a specific path and this cycle has more than two elements, the path is duplicated and the cycle is added to the duplicate. This step is repeated according to the number of cycle iterations desired (line 9).

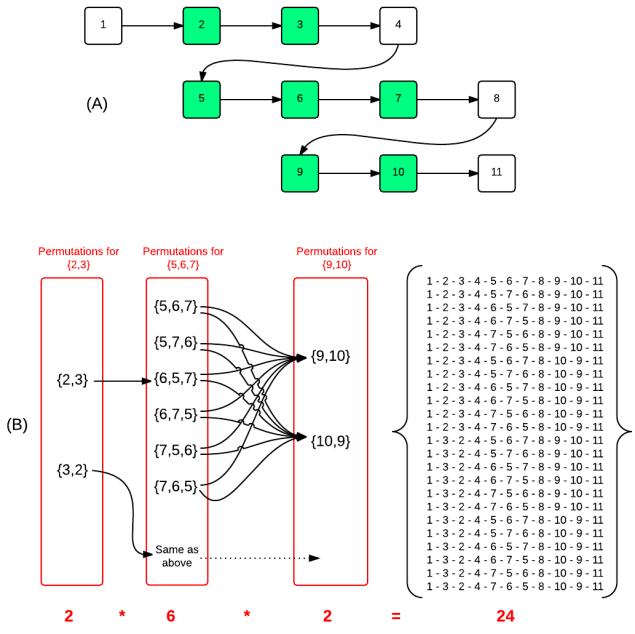

Figure 4: (A) Path with multiple out of order sets; (B) The fundamental counting principle.

The cycles are processed only if the cycle sequences contain more than two elements (line 6). This rule is justified since the algorithm detailed on Section 3.1 already handles cycle with two elements. At that point, several cycles were added between the out-of-order elements to simulate the specified behavior. So these short cycles are not processed again during this step.

When a cycle element is found in a path, the cycle is added into the original path and also to the respective variations related to the out-of-order and optional elements. The *for* statement on line 7 is responsible for distributing the cycles within all paths.

4. EVALUATION

An empirical study was conducted to analyze the approach described in this paper. A web application based on renowned technologies such as AJAX was used to evaluate the effectiveness of the test cases generation. The goal of this study was to measure if the test cases created by UsaTasker++ are able to cover the application's functional requirements.

The System Under Test (SUT) was a financial web site that provides loans with low interest rates and a covenant in which the loan needs to be spent in a restricted set of vehicles of an specific automaker. Phase 1 of the experiment was to define the tasks (use cases) provided by the SUT and the functional requirements (FRs) related to each task.

Due to space restrictions, we will focus our attention on the task "*buy a vehicle*". The FRs for this task are as follows:

I. User can submit an offer for one vehicle at a time, but several offers can be submitted;

II. *Name* and *address* are mandatory fields, while *address complement* is optional;

Algorithm 3: Algorithm *ProcessCycles*

Input: from-to events hash-map C, list of all paths P, number of cycle iterations NC
Output: list of paths with cycle CL

```
1  s foreach event fromE in C do
2      toE ← C(fromE) ;
3      foreach event sequence seq in P do
4          if fromE and toE are in seq then
5              cycle ← sub-list of seq(toE, fromE) ;
6              if cycle contains more than 2 events and not
               already processed then
7                  foreach event sequence seq2 in P do
8                      if fromE is in seq2 then
9                          for i ← 0 to NC do
10                             newSeq ← seq2(0, fromE) ;
11                             for rep ← 0 to i do
12                                 newSeq ← newSeq + cycle
                                   ;
13                             newSeq ←
                               newSeq + seq2(fromE, end) ;
14                             add newSeq to CL ;
15 return CL ;
```

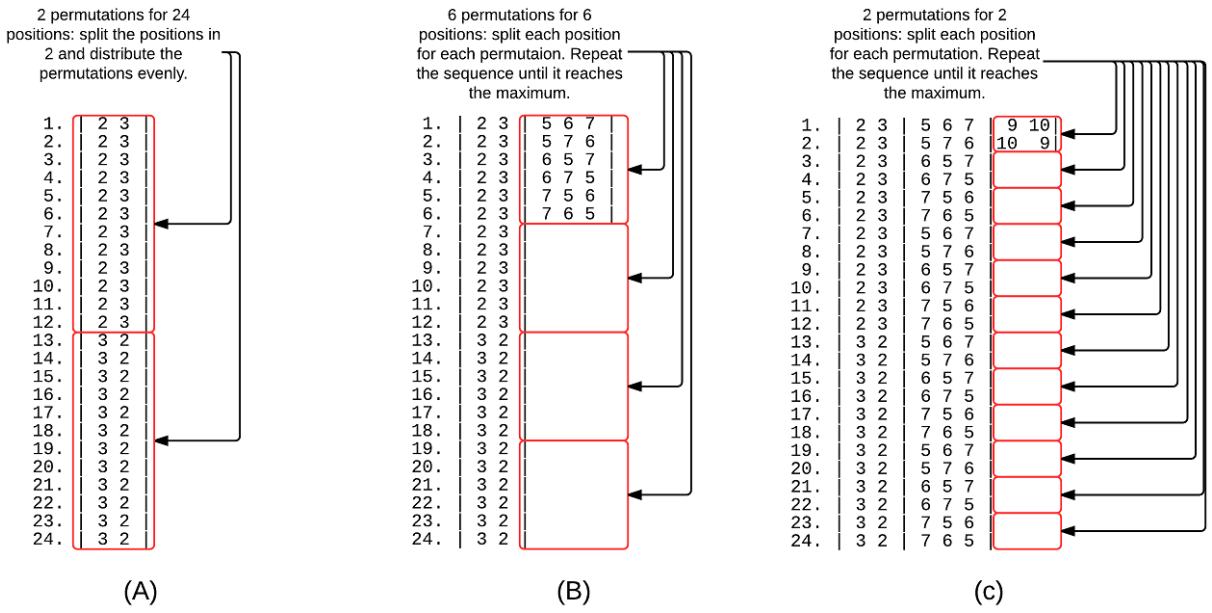

Figure 5: Group permutations: (A) First group; (B) Second group and (C) Third group.

III. User must accept the "Terms of Use". Otherwise, an error is shown.

IV. The fields in the page can be filled in any order.

In the second phase, the basic flow for the task was recorded. Within this phase, the events were characterized as Optional, Out-of-Order and/or Cyclic, based on the FRs gathered on the previous phase. The graph in Figure 6 represents the model for the task. According to the FRs, all the three characteristics were explored.

Note that the steps defined in this phase are tightly related to the success of the test case coverage. The Test Analyst should prepare the flow accordingly so that all FRs can be explored. For experimental reasons, this paper will analyze the shortened flow shown on Figure 6 and later will compare it with a better approach.

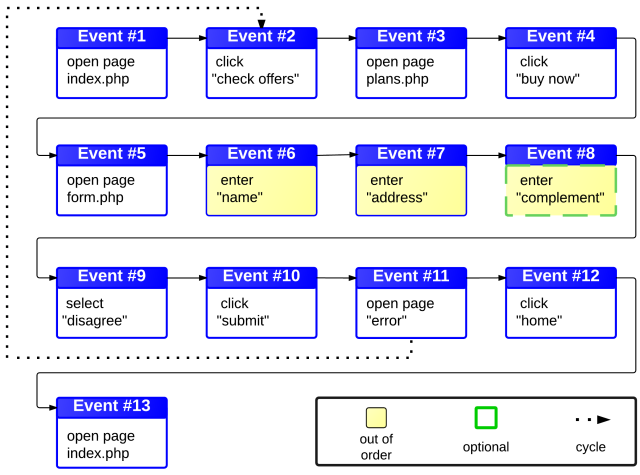

Figure 6: The task "Buy a vehicle" expressed as a sequence of events (shortened flow).

The last phase of our experiment was to check if the generated test cases were able to cover all the application's FRs, as well as all the possible sequence of events. For the graph depicted in Figure 6, UsaTasker++ was able to generate 72 test cases in total, 8 without cycles (referred in this paper as *Simple TCs*) and 64 tests with cycles (*Cycled TCs*).

As indicated on the first line of Table 1, 2 test cases are related to the optional element – one sequence with the element and another without it – and 6 test cases are related to the out-of-order elements – derived from the permutation of 3. Each one of these 8 Simple TCs contains a different cycle sequence in it. Therefore, the 64 Cycled TCs are originated from the operation that relates each cycle with the Simple TCs – 8 cycle sequences X 8 Simple TCs.

In the second line of Table 1, we executed the same experiment, but considering 3 cycle iterations. This time, we had 3 times the 64 Cycled TCs, since we created a sequence with one cycle iteration, another with two iterations and another with three iterations. The total test cases were 8 Simple TCs and 192 Cycled TCs.

According to the permutation rules described on Section 3, as well as the logic defined for the optional and cycle elements, this experiment was successful in creating all possible test cases. However, the generated test cases were not capable of covering FR I and III. Although many Cycled TCs were generated, none of them submitted a real offer as it did not accept the "Terms of Use". All the TCs were manipulating the GUI elements repeatedly, verifying the system behavior on negative scenarios. It was clear that the short set of actions recorded by the Test Analyst did not go through all the possibilities of the GUI, leaving some scenarios untouched.

After re-validating the base scenario, an extended approach was determined, as depicted in Figure 7. This time, both options were recorded ("Agree" and "Disagree") on the same test sequence. As a result, the number of test cases increased to 4160. Most important, the generated test cases were now capable of covering all FRs of the SUT.

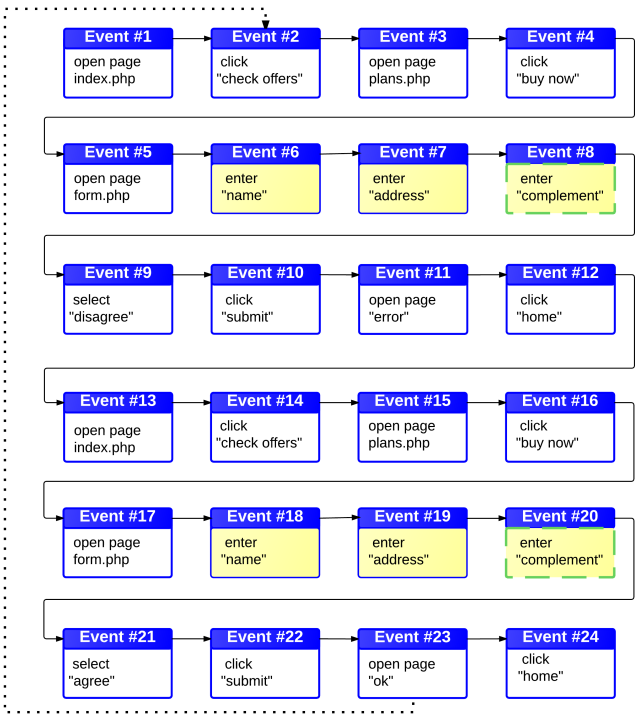

Figure 7: The task "Buy a vehicle" expressed as a sequence of events (extended flow).

The 64 Simple TCs created during this validation (third and fourth lines of Table 1) successively tests all the possible positive and negative scenarios, which are based on the "Terms of Use" acceptance. Later, each cycle sequence of those TCs are distributed in each Simple TC, generating 4096 Cycled TCs (64 cycles *X* 64 Simple TCs). When setting the cycle iterations to 3, the number of Cycled TCs is triplicated, creating a total of 12352 unique test scenarios.

Table 1: Test Results

Approach	CI	Ooo	Opt	Cy	Total
Shortened	1	6	2	64	72
Shortened	3	6	2	192	200
Extended	1	60	4	4096	4160
Extended	3	60	4	12288	12352

CI: cycle iterations; Ooo: out-of-order; Opt: optional; Cy: cyles.

As observed on Table 1, the number of test cases generated is proportional to the base scenario expansiveness. The results, thus, highlight the significant amount of unique scenarios and the ability to generate all the possible sequences. Nevertheless, it also stresses the application's functional requirements, demonstrating that our technique is successful in achieving a complete functional coverage.

5. RELATED WORK

The work by Ricca and Tonella [13] is one of the first relevant efforts to generate test cases for web applications. They use a graph abstraction to represent the application, where nodes represent pages, forms and frames, while edges represent the relations among nodes, such as link, submit and include. Nodes and links are then used as coverage criteria to derive test cases.

Besides Ricca and Tonella, many other authors have proposed model-based approaches for generating test cases. The work developed by Lucca et al. [10], for instance, abstracts a web application using an object-oriented model and present a decision-table-based method for generating test cases.

As far as model-based testing is concerned, it is also important to cite the works that tries to model a web application using state diagrams. Andrews et al. [1] use finite state machines (FSM) to model the behavior of a web application. According to their model, nodes in the FSM represent web pages and software modules, and edges represent transitions among the pages and modules. Thummalapenta et al. [15] also exploits state transition diagrams (STD), which are created by crawling the application GUI. They developed, however, a set of algorithms that process the STD identifying rule relevant abstract paths. They show that their approach is effective for identifying paths that covers business rules. Therefore, their work is more closely related to ours, as it is also targeted to cover functional requirements.

The literature also present some efforts to generate test cases without relying on application models. These efforts fall basically in two main categories: unguided crawling and user-session approaches. Unguided crawling techniques exhaustively explore the pages of a web application, generating a navigational graph that can be used to test the application. The first crawling approaches explored the regular hyperlinks inside of each page to perform the crawl [3, 18]. With the advent and widespread of Javascript and Ajax technologies, new approaches have been proposed to handle state changes that do not involve a page reload [9, 11]. Despite of being able to generate a huge volume of test cases, these approaches are not effective in covering the functional requirements of applications.

User-session approaches generate test cases using logged data gathered during application usage in the field [6, 14]. The advantage of recording user-sessions is the fact that users are not being supervised while they perform tasks and thus unexpected ways of executing a given task may occur, which is important for application testing. On the other hand, it is hard to assess the coverage of resulting tests, as it depends on what had been recorded. Moreover, even when the application's main tasks are recorded, it is not possible to assure that users have performed a task using all the possible navigational paths provided by the application – most likely, they did not. Finding those paths is where UsaTasker++ excels, providing a solution that leverages user-session approaches. Another advantage of our solution is the fact that we do not have to deploy an application for real usage in order to gather logging data. UsaTasker++ makes it possible to test the application during its development, avoiding the deployment of defective applications.

6. CONCLUSION

Different approaches have been proposed for web application testing. None of them, however, are effective in order to cover the functional requirements of applications. To tackle this problem, we presented a task-based approach for functional testing. As our test case generation procedure is guided by pre-recorded tasks, the resulting test suites tend to present high coverage of functional requirements: coverage may reach 100% when recorded tasks are well chosen.

Another advantage of our approach is that instead of relying on complex application models or costly unguided crawling algorithms, we exploit task-based data that can be easily recorded by simply interacting with the web application GUI.

The novelty of our solution is a set of algorithms which exploit the features of a navigational model that takes into consideration the different possibilities in which a task may be performed. To the best of our knowledge, no other work exploits these navigational features in order to generate test cases.

As important as generating test cases is evaluating their results. This is the task of the oracles. In most of the cases, oracles only determine whether a test case has passed or failed. As future work, we plan to develop oracles that, in case of failure, identify the GUI element where the failure occurred. We advocate that this feature is able to shorten the time to fix a defect, improving software productivity.

7. BIOGRAPHIES

Flávio Rezende de Jesus is a MSc student in Computer Science at Federal University of Itajubá. His interests are related to Software Engineering, focussed on testing methodologies and relationship algorithms.

Laércio Augusto Baldochi Jr. is an assistant professor of Computer Science at Federal University of Itajubá, where he leads the Usability and Interactive Media Lab. He received his PhD in Computer Science from University of São Paulo. His research interests include web engineering, human-computer interaction and semantic web.

Leandro Guarino de Vasconcelos is a PhD candidate in Applied Computing at Brazilian Institute for Space Research. His interests are related to Human-Computer Interaction, focussed on remote usability evaluation and user behavior analysis.

8. REFERENCES

[1] A. A. Andrews, J. Offutt, and R. T. Alexander. Testing web applications by modeling with fsms. *Software and Systems Modeling*, 4:326–345, 2005.

[2] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 571–580, New York, NY, USA, 2011. ACM.

[3] M. Benedikt, J. Freire, and P. Godefroid. Veriweb: Automatically testing dynamic web sites. In *In Proc. of 11th Int. World Wide Web Conference – WWW 2002*, 2002.

[4] V. Dallmeier, M. Burger, T. Orth, and A. Zeller. Webmate: A tool for testing web 2.0 applications. In *Proc. of the Workshop on JavaScript Tools*, JSTools '12, pages 11–15, New York, NY, USA, 2012. ACM.

[5] S. Dogan, A. Betin-Can, and V. Garousi. Web application testing: A systematic literature review. *Journal of Systems and Software*, 91(0):174 – 201, 2014.

[6] S. Elbaum, G. Rothermel, S. Karre, and M. F. II. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, 2005.

[7] B. R. Heap. Permutations by interchanges. *The Computer Journal*, 6(3):293–298, 1963.

[8] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 67–77, New York, NY, USA, 2013. ACM.

[9] G. Le Breton, F. Maronnaud, and S. Hallé. Automated exploration and analysis of ajax web applications with webmole. In *Proc. of the 22nd Int. Conf. on World Wide Web Companion*, WWW '13 Companion, pages 245–248, Geneva, Switzerland, 2013. WWW Conferences Steering Committee.

[10] G. D. Lucca, A. Fasolino, and F. Faralli. Testing web applications. In *Proc. of the Int. Conf. on Software Maintenance (ICSM'02)*, ICSM '02, pages 310–319, Washington, DC, USA, 2002. IEEE Computer Society.

[11] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. Web*, 6(1):3:1–3:30, mar 2012.

[12] A. Nederlof, A. Mesbah, and A. v. Deursen. Software engineering for the web: The state of the practice. In *Companion Proc. 36th Int. Conf. on Software Engineering*, ICSE Companion 2014, pages 4–13, New York, NY, USA, 2014. ACM.

[13] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proc. of the 23rd Int. Conf. on Software Engineering*, ICSE '01, pages 25–34, Washington, DC, USA, 2001. IEEE Computer Society.

[14] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *Proc. of the 20th IEEE/ACM Int. Conf. on Automated Software Engineering*, ASE '05, pages 253–262, New York, NY, USA, 2005. ACM.

[15] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra. Guided test generation for web applications. In *Proc. of the 2013 Int. Conf. on Software Engineering*, ICSE '13, pages 162–171, Piscataway, NJ, USA, 2013. IEEE Press.

[16] L. G. Vasconcelos and L. A. Baldochi. Towards an automatic evaluation of web applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 709–716, New York, NY, USA, 2012. ACM.

[17] L. G. Vasconcelos and L. A. Baldochi, Jr. Usatasker: A task definition tool for supporting the usability evaluation of web applications. In *Proceedings of the IADIS International Conference WWW/Internet*, ICWI 2012, pages 307–314, 2012.

[18] W. Wang, Y. Lei, S. Sampath, R. Kacker, R. Kuhn, and J. Lawrence. A combinatorial approach to building navigation graphs for dynamic web applications. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 211–220, Sept 2009.