Test Scenario Generation for Web Application Based on Past Test Artifacts

Rogene LACANIENTA†, Nonmember, Shingo TAKADA†(a), Member, Haruto TANNO††, and Morihide OINUMA††, Nonmembers

SUMMARY For the past couple of decades, the usage of the Web as a platform for deploying software products has become incredibly popular. Web applications became more prevalent, as well as more complex. Countless Web applications have already been designed, developed, tested, and deployed on the Internet. However, it is noticeable that many common functionalities are present among these vast number of applications. This paper proposes an approach based on a database containing information from previous test artifacts. The information is used to generate test scenarios for Web applications under test. We have developed a tool based on our proposed approach, with the aim of reducing the effort required from software test engineers and professionals during the test planning and creation stage of software engineering. We evaluated our approach from three viewpoints: comparison between our approach and manual generation, qualitative evaluation by professional software engineers, and comparison between our approach and two open-source tools.

key words: software testing, scenario generation, knowledge-based engineering, Web applications

1. Introduction

Software testing is a critical part of software development, but it can be very costly and time-consuming. Many methods have been proposed to automate various facets of software testing so that the testing cost can be reduced [1]–[7]. However, they only consider information that is available from the software-under-test.

Various software artifacts are developed during software development. These include requirement documents, design documents, source code, test scenarios, and test cases. There have been numerous work concerned with reusing such artifacts, but they have focused especially on the reuse of code [8]–[11]. We consider reuse of information concerned with testing.

We take a knowledge-based approach that taps into the potential of already-available, high-quality software artifacts (e.g., design documents, source code, test scenarios, and test cases) to produce robust test scenarios, which can be used to perform validation testing on a software-under-test. Kaner described a test scenario as a test based on how the program is used [12]. In this paper, we focus on Web applications and their Web pages, and view a test scenario as a sequence of steps that are taken to transition from one Web page to another. For example, a test scenario for registration may include steps such as input a username, input a password, and submit the information. The key difference between our test scenario and conventional test cases is that a test case will have the actual test data itself, while our test scenario will have the type of test data. For example, a test case may have “john@xyz.com” as the value of an e-mail address, while it may be “NORMAL” (indicating a valid e-mail address) in our test scenario. We will explain our test scenario further in Sect. 3.1.

Our approach is based on a database containing test scenarios that are collected from previous applications. A test engineer creates a base scenario of the software-under-test and uses that to query for related test scenarios.

The main contributions of our work are:

1. An approach that systematically reuses testing information, specifically test scenarios of previous tests.
2. An implementation of our approach.
3. An evaluation that is done from various perspectives.

This paper is an extended version of [13] where we first introduced our approach and conducted a case study to compare the performance of our approach to that of professional software test engineers. For this paper, we added more comprehensive evaluations for our approach by comparing the output of our tool to the output of two other open-source tools. We also added a qualitative study based on the opinions of professional software engineers.

The rest of this paper first starts with a discussion of related work. Section 3 then describes our approach. We give an overview and then give details on the test scenario database as well as the querying. Section 4 describes an evaluation using professional test engineers. Section 5 compares our tool with two open-source tools. Finally, Sect. 6 makes concluding remarks.

2. Related Work

There are many work on generating test scenarios and test cases. Xu, et al used adaptive agents that traverse a UML diagram representing a test scenario [1]. During traversal, the agents perform manipulation of the nodes based on a fixed algorithm. Andrews, et al. proposed a method to use finite state machines to model Web applications, which are then used with corresponding constraints in order to output test
cases with minimal state space explosion [2]. Fujiwara, et al. introduced an approach that analyzes UML diagrams and uses a combination of Object Constraint Language (OCL) and Satisfiability Modulo Theories (SMT) solver to produce test cases [3]. Artzi, et al. used symbolic execution to dynamically create test cases and find bugs [4]. Halfond, et al. conducted static analysis to discover Web application interfaces and help generate test inputs [5]. Although not intended for Web applications, Pacheco, et al. developed Randoop which conducts random testing with feedback [6], and Gross, et al. proposed a genetic algorithm-based approach based on GUI to generate a high coverage test suite while avoiding false failures [7].

Works such as these effectively increase the coverage of test cases and/or their bug detection ability. But, they fail to consider the knowledge available from other possible sources of test information — that is, already deployed Web systems and their corresponding software artifacts. We consider how such information can be used to generate useful test scenarios.

There are some existing work that uses prior information from previously developed software to generate test cases. Landhausser, et al. [14] proposed an approach that clones test cases basing it on the hypothesis that common test cases can be used for software that have very similar functionality. Recently, Janjic, et al. [15] proposed developing test cases by searching for prior code with similar classes and methods, along with their corresponding test cases. Both work rely on similar source code as their knowledge base to generate code-level unit tests. In our approach, we aim to propose a more general-purpose test case generation system that leverages on the higher level of information that are scenario steps and generates functional-level test scenarios.

### 3. Knowledge-Based Test Scenario Generation

In this section, we first describe the test scenario used in our approach. We next show an overview of our tool, and then the entries in our database. Then, we describe how scenarios are generated and ranked. Finally, we describe our implementation.

#### 3.1 Test Scenario

A test scenario is basically a sequence of steps that are taken to transition from one Web page to another. It is similar to a conventional test case, except that a test case will have the actual test data for each input, while our test scenario will have the type of test data.

We visualize a scenario as a diagram similar to a UML activity diagram. The reason for using a diagram that closely resembles a UML activity diagram is because we believe that it will be easier for the test engineer to draw diagrams that are similar to what they know than something that is completely new.

There are three basic types of nodes:

1. **Processing** is a node indicating an internal computation in the Web page. This can come in the form of Javascript method or any other dynamic processing or verification procedure that can happen within a Web page. Note that in many cases, the “Processing” within the Web page is simple, and complex processing is done at the Web server.
2. **Input** is a node indicating the information the Web page requires from the user.
3. **Output** is a node indicating some type of output by the Web page that gives feedback to the user. This can be in the form of pop-up alert box or dynamic textual/graphical changes within the same Web page. Output nodes can then be utilized by the test engineer to indicate the expected results for a particular test scenario.

Of course, there are also the start and end nodes.

Figure 1 shows an example of a test scenario for the Registration process. It has 8 nodes (including the start and end nodes). The second node is a Processing node where a Web page is loaded. The following four nodes are all Input nodes, which require user input. Each Input node has an at-
tribute and a value associated with it. For example, the third node has “USER ID” as the attribute and “NORMAL” as the value. “NORMAL” denotes the type of value, and indicates that the actual value used in testing should be a valid one. The last Processing node expresses the action of submitting information to the Web server, where the actual computation takes place. Finally, since this example does not have an Output node, no explicit expected result has been specified. The test engineer may add an Output node just before the end node if he/she wants to specify an expected result.

3.2 Tool Overview

In our approach, the test engineer sends a query to a database containing test scenarios to obtain related test scenarios that can be used for the current software-under-test. The basic steps are as follows (Fig. 2):

1. The test engineer inputs a base scenario using the Scenario Editing Tool.
2. The base scenario is analyzed by the Scenario Analyzer.
3. Each node in the base scenario is used to query for related nodes in the Scenario Node Database. A dictionary is used to expand the range of possible matches.
4. The Related Scenario Generator constructs combinations of the nodes to generate scenarios that are related to the base scenario.
5. The generated related scenarios are ranked and returned to the test engineer.
6. The Scenario Node Database is updated based on the generated related scenarios.

3.3 Database

Each entry in the database corresponds to a node rather than a test scenario. The following information are included in every entry:

- **Process Type**: The pre-defined type of operation where the node appeared. Our current implementation is limited to “Registration”, “Log In”, “Search,” and “New Content”, but we can extend this in the future.
- **Node Type**: Either Processing, Input, or Output. This paper mainly focuses on Input, as they make up the majority of nodes.
- **Attribute Name**: The name of the attribute.
- **Node Value**: The value of the node.
- **Frequency**: The number of times a node occurs in past Web applications. This value plays an important role when ranking the generated related scenarios.

3.4 Node Matching

A related scenario is a scenario where one node in the base scenario has been replaced with one matching node from the database. Ideally, we would not limit ourselves to one node and allow multiple nodes to be replaced. However, this may result in combinatorial explosion. Thus, we only consider related scenarios that differ from a base scenario by only one node.

For each node in the base scenario, the Scenario Analyzer searches for matching nodes in the database. A match occurs when the node information and the database entry information match; more concretely, when the Process Type, Node Type, and Attribute Name all match. When the Node Type is Input, the Scenario Analyzer searches for Input type nodes in the database with a matching attribute. There are two types of matches for attributes: (1) exact match and (2) similar match. An exact match literally means that the attribute names for the nodes are the same. For example, an Input node in the database with “USERNAME” as an attribute can be considered as similar to a “USER ID” Input node. A dictionary is used to determine if two attribute names can be considered to be synonyms.

Note that the node matching focused only on the Attribute Name even though there are other information such as the Node Value. As a result, matching nodes will include those with different Node Value. For example, the result of searching for an Input node with “USER ID” as an attribute and “NORMAL” value (henceforth denoted as “USER ID=NORMAL”) may result in matches such as
“USER ID=TAKEN”, “USER ID=DOES NOT EXIST”, etc. These are the nodes that are central to generating the related scenarios.

Also note that each node will have a frequency value, which will be used for ranking the final related scenarios. In the case of a similar node, the original frequency value is weighted to take into consideration the factor that this particular node is not an exact match, but a close one. For this paper, we chose the value 0.75. This value was chosen based on a preliminary experiment with a very small dataset. We compared the results of various values to determine the best value for the dataset. However, the best value may change for different datasets, and thus further evaluation needs to be done.

Finally, note that the order of the nodes is not considered when calculating the similarity as we consider each node individually.

The resulting matching nodes are now sent to the Related Scenario Generator. The Related Scenario Generator takes each node in the base scenario, and replaces it with a matching node to form a related scenario. Note that each related scenario will only have one matching node; all other nodes in the related scenario are the same as the base scenario. Figure 3 shows an example of two related scenarios given the base scenario in Fig. 1. In Fig. 3(a), the value for the “USER ID” node is changed from “NORMAL” to “TAKEN”, meaning that the username has already been taken by someone and cannot be used for a new member. In Fig. 3(b), it is changed to “LONG”, meaning that the username is too long and needs to be shortened.

The Related Scenario Generator also attaches a score to each related scenario. Scoring will be described in the next subsection.

3.5 Scoring and Ranking

Our tool ranks the generated scenarios based on the frequency of each node, more specifically the number of times the node was used to test past Web applications. The frequency value of each node is available in the database (Sect. 3.3).

The ranking is done by first computing the score of each generated scenario. The following formula is used for this.

$$score = \alpha \times freq$$

where $\alpha$ is a coefficient taking a value between 0 to 1, and $freq$ is the frequency value of the node that was replaced.

When the match is an exact match, the value of $\alpha$ is set to 1. If it is a similar match using a dictionary, we have set the value to 0.75. The ranking of the related scenarios are done based on the above score. A higher score basically means that the replacement node occurred more often in many past Web applications, and can thus be considered to have higher priority.

3.6 Tool

We implemented our approach using UMLet Tool [16], an open-source Java-based application for creating UML diagrams. Using MySQL, we constructed the database for knowledge obtained from previous Web applications.

Figure 4 shows the main screen of our tool, with an example scenario. On the left side of the tool is the canvas for creating a base scenario for the software-under-test. Just like in other UML editors, the right-hand toolbar contains the basic elements that can be used to draw the base scenario. As seen in the figure, the resulting scenario closely resembles a UML activity diagram. When “Generate Related Scenario” button is pressed, the related scenarios are generated. The resulting related scenarios are shown in separate canvases on the left side of the tool. Test engineers can choose a tab to display the corresponding related scenario.

The “Process Scenario” button is for adding a new scenario to the database. The developer first draws the scenario on the left side of the tool before pressing the button. The system automatically parses the scenario for each state and populates the database accordingly.

The initial database population needs to be conducted by choosing programs with “good” documentations. The documentations and the source code are used to create the scenarios for the Web pages, which are registered in the database. A scenario node is either input into the database as a new node if it does not exist, or the frequency is updated if the same entry already exists.

The initial dictionary creation can be done in parallel with the database population. The database populator
helps determine synonymous terms by selecting possible synonyms for attribute names of nodes if it has not been input yet.

The resulting dictionary as well as the scenario node database should be reviewed, and any mistakes should be corrected.

4. Evaluation with Professional Test Engineers

We evaluate our approach from three perspectives: (1) comparison of manual vs. automatic test scenario generation, (2) qualitative survey of the approach, and (3) comparison of our approach to existing tools. The first two perspectives will be described in this section, while the third will be given in the next section. Before we describe the evaluation, we first describe how we populated the database and created the dictionary that was used in our evaluation.

4.1 Database Population and Dictionary Creation

We populated our database and created the dictionary, as described in Sect. 3.6. We first surveyed available open-source programs with “good” documentations. We concentrated on four types of processes common to many Web applications: 1) *Registration* of a new user, 2) *Login* where a user inputs his/her credentials to enter the Web application, 3) *Search* where an element of the Web application can be searched, and 4) *New Content* where a new entry in the Web application’s database can be input. We also concentrated on these six types of Web applications: a) CMS systems, b) Wiki systems, c) Forum applications, d) Blogging applications, e) Shopping and Hotel Management Systems, and f) Online Educational Systems. This resulted in a set of twelve open-source Web applications.

Then we manually added scenario nodes to the database and created the dictionary in parallel. This resulted in a knowledge database containing about 900 unique scenario steps and 180 dictionary terms.

The cost of constructing the database took a total of about 30 man-hours. About 25 man-hours were spent on analyzing, encoding, and saving the Web application scenarios. Another 5 man-hours were spent performing post-processing of the data in order to ensure accuracy and correctness. This mainly involved correcting typographical errors in the data entries, as well as consolidating duplicated entries that resulted from these corrections. Since this was done manually, there is always the possibility of an error remaining in the database, but no such errors were found during the evaluations we conducted.
4.2 Manual vs Automatic Test Scenario Generation

We first describe a comparative study between manual and automatic test scenario generation. Further details of this comparative study can be found in our previous paper [13].

4.2.1 Evaluation Approach

For this evaluation, we used two open-source Web applications: Tapestry Hotel Booking System (THBS) [17] and Online Learning and Training System (OLAT) [18]. We compared the output of three professional engineers with at least 5 years of working experience versus the output of our tool, as used by a student. We focused on comparing three factors: Time, Quantity, and Quality.

The basic procedure for the experiment is as follows:

1. The engineers were given the documentation, source code, and running instances of the Web applications to be tested. The engineers were asked to construct test scenarios for these Web applications using the method and output they were most comfortable with.
2. The student was given running instances of the Web application and was asked to construct the base scenario for them using the proposed tool. The student then generated the related scenarios using our tool.
3. The two sets of scenarios were compared.

4.2.2 Evaluation Results

Table 1 shows the time taken to generate the scenarios. The engineers took more than twice the amount of time to produce their scenarios compared to the student using the tool. The main reason is that the engineers needed to get familiar with the software-under-test first, and then manually write the scenarios. The student, on the other hand, needed to understand only the basics of the Web application so that he could encode the Web pages’ input fields into the tool and execute the scenario generation (which took about two minutes).

Table 1 Time to generate scenarios.

|          | THBS     | OLAT     |
|----------|----------|----------|
| Engineers| about 60 minutes | about 90 minutes |
| Student+Tool| 25 minutes   | 27 minutes |

Table 2 Number of generated scenarios.

|          | T: Scenarios generated by tool | E: Scenarios generated by engineer | T ∧ E: Scenarios generated by both tool and engineer | E ∧ ¬T: Scenarios generated by engineer but not tool | T ∧ ¬E: Scenarios generated by tool but not engineer |
|----------|-------------------------------|-----------------------------------|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|
| THBS     | 58                            | 30                                | 28                                            | 2                                             | 30                                            |
| OLAT     | 81                            | 44                                | 32                                            | 12                                            | 49                                            |
| Total    | 139                           | 74                                | 60                                            | 14                                            | 79                                            |

Table 3 Analysis of generated scenario usefulness.

|          | T ∧ E: Useful | Noise |
|----------|---------------|-------|
| THBS     | 30            | 13    | 17   |
| OLAT     | 49            | 29    | 20   |
| Total    | 79            | 42    | 37   |

correspond to any database or dictionary entry. For example, an “Enter Text Display” field asks for a security text (a captcha). In this case, the knowledge stored in the database and dictionary did not match the entered information, even if, technically, relevant related scenario nodes were available. If the attribute name was set to “captcha”, “security text”, or any similar entry, then relevant nodes would have matched and a related scenario could have been generated. This is a limitation of our approach that needs further work.

We then further checked the 79 scenarios that our tool generated but were not created by the engineers (“T ∧ E” column of Table 2). We found that 53% of such scenarios were useful for testing the software-under-test while 47% were not (noise) (Table 3).

Noisy scenarios are generated related scenarios that are not useful for testing the software-under-test. For example, a value of “false” for an input node “Gender” can be considered as noise since this has no real meaning as a test scenario. The most common reason for the generation of noisy scenarios is the fact that our approach does not consider input types (e.g. text field, radio button, check box, etc.) For example, when populating our database, if we encounter an input with the attribute name “Description” that is a check box, this would be added to our database with “true” and “false” values. However, when using our tool, suppose an input with the same attribute name “Description” is encountered but as a text field. Our tool will then generate test scenarios with “true” and “false” values. These scenarios are deemed noisy as they cannot be used to test the software-under-test.

Finally, we analyzed the noisy scenarios to see where they appeared in terms of the ranking. Specifically, we took the top 75% of the tool generated scenarios in terms of ranking and checked the breakdown (Table 4). The total number of noisy scenario in the top 75% ranked scenarios was 12, which is 32% (=12/37) of all noisy scenarios. In other words, the 25% lowest ranked scenarios contained 68% of the noise, while it only contained 16% of the useful ones. This indicates that the noise mostly appears in the lower ranked scenarios, so by tuning the cutoff point of the sce-
Table 4  Usefulness of top 75% scenarios.

|      | T \&\neg E | Useful | Noise |
|------|------------|--------|-------|
| THBS | 17         | 11     | 6     |
| OLAI | 30         | 24     | 6     |
| Total| 47         | 35     | 12    |

scenario ranking, we can eliminate most of the noise from the output. Note though that there is a tradeoff since eliminating noise based on ranking also means that some of the useful scenarios will also be eliminated.

4.2.3 Discussion

Our current approach is limited in the sense that it cannot automatically judge if a generated scenario is useful or noise. It is left to the expert judgment of the test engineer to decide, depending on the specifics of the software-under-test. However, it is worth emphasizing that we found that noisy scenarios were generally ranked low. This may act as a guide to future work, where we plan to study how to automatically detect noisy scenarios. Also, as we noted above, one cause of the noisy scenarios is the lack of considering input types. Thus we also plan to incorporate input type in our future work to prevent some of the noisy scenarios.

Our approach is also limited in the sense that it can still miss some scenarios that professional test engineers can produce. While this accounts to a relatively small percentage and can further be reduced by future improvements in our methodology, it still is a current limitation. In such cases, the missed scenarios can hopefully be noticed via traditional test specification review procedures done in professional software development. Thus, it is still necessary for a test team to conduct reviews for the output of our automated approach, as dictated by common industry best practices.

In sum, our initial evaluation indicated the following:

- Our approach can generate more scenarios quickly.
- Our approach can generate scenarios that engineers miss.
- Noisy scenarios are generated due mainly to database issues.

Also, upon the result of this round of evaluation, we were able to gather that, while both the tool’s output and the engineers’ output initially contain duplicates, manual analysis of our database can be executed in order to reduce the number of duplicates, minimizing the amount of post-process analysis needed for the tool’s output. This particular action step was made before proceeding with the following evaluations.

4.3 Qualitative Survey

To get an overview of how useful our approach is in the real-world software engineering context, we conducted a survey among four software engineers from three different software companies and with at least 3 years of software engineering experience who are adept at software design, development, and testing. This survey was divided into three segments. The first one involves the discussion of the underlying principles of our approach as well as the demonstration of the developed tool with the explanation of how the current database was populated.

Afterwards, the engineers were given ample time to try out the tool with various random scenarios they can think of. This is to give them a basic feel on how the tool works, as well as to gauge the quality of the output that the database is able to produce. The engineers were then asked to evaluate the approach and the tool based on the following three questions:

1. How easy or difficult is it to understand the concept and usage of the tool?
   (1- very difficult 2- difficult 3- moderate 4- easy 5- very easy)
2. Do you think this tool will be useful in your work?
   (1- not at all 2- maybe not useful 3- not sure 4- useful 5- very useful)
3. How easy or difficult do you think it is to adapt the tool and approach to industry-level software testing?
   (1- very difficult 2- difficult 3- moderate 4- easy 5- very easy)

Figure 5 shows the results based on the high, low, and average scores. The lowest score for each of the questions was 3 or greater. We especially focus on the second question, where all responses were either 4 or 5. This indicates the potential of our tool to be used by the engineers.

Professional opinions were also solicited from the survey subjects. Overall, the engineers gave a positive review of the approach. The consensus opinions were as follows:

- The idea is simple to understand. The creation of base scenarios is straightforward and simple from a software engineer’s point of view.
• The tool covers many of the basic test scenarios. This reduces the burden from test engineers, giving them more time to think of more special and system-specific test scenarios.
• The output seems usable but may involve manual translation to test cases, depending on how this will be processed.
• The strength and effectiveness of the tool relies heavily on the database. More input means more useful scenarios can be generated.

Overall, the survey gathered some positive feedback. However, this is a very limited sampling population and cannot be used to produce an overall conclusion in terms of the usefulness of the approach in the industry. However, the result suggests that our approach will be useful to professional software engineers.

5. Comparison with Other Tools

To further evaluate the effectiveness of our approach, we compared our tool with two existing open-source tools.

5.1 Test Generation Tools

The two tools were Spawner Data Generator [19] and Test Case Generator (TCG) [20].

Spawner Data Generator is a generic generator of test cases. It supports the generation of random test case data, given a set of parameters. It is capable of generating a large amount of strings, numerals, time, human names, network IDs, etc. Spawner accepts input fields as parameters for data generation. Here, the user can specify some restrictions for the data of each field. For example, a range can be specified for numeric values, length can be indicated for strings, and gender can be chosen for human names. The number of randomly generated test scenarios can be defined. In the case of this study, 100 test cases were produced.

Test Case Generator (TCG) is a tool for generating test cases based on equivalence class analysis and combination of states. Given a set of input fields, it uses an exhaustive combinatorial algorithm to extract all the possible combinations of input values to produce test cases. TCG accepts input fields where the user can define possible input types (valid, invalid, etc) and some sample data and use this information to produce an exhaustive array of possible combinations.

These two tools were chosen based on the similarity of the input/output. For input, the two tools both need the attribute name and some possible attribute values in order to generate test artifacts. Furthermore, the output, i.e., the test cases, produced by both tools each contain a set of input values for all the fields of the target Web page. We can fairly easily compare the output test values of the tools to the attribute values present in the related test scenarios produced by our own tool. For example, our tool’s output value “no @ sign” for an email address attribute is equivalent to the other tool’s “student_domain.edu” test case data. Thus, although we had to manually check the outputs (details in the next subsection), we were able to treat each test case as comparable to our tool’s produced test scenarios.

5.2 Test Scenario Comparison

For this evaluation, we selected the Cyclos Project [21], an open source Web application for banking. Figure 6 shows the target Web page. We focused on three of its available Web pages, 1) “New Advertisement”, 2) “Payment”, and 3) “Registration”.

For the target Web page, each tool (Spawner, TCG, and our approach) generated test scenario/data independently. This covers a total of six input fields. Table 5 shows the total number of test cases/scenarios produced by each tool. Note that for our proposed approach, the total number of test scenarios include the base scenarios that were input by the test engineer.

It is worth noting that although Spawner generated 300 test cases (100 for each Web page), it can actually generate more. But, for this particular study, since the data included in these extra cases are similar in nature, we deemed that using more than 100 was unnecessary. Since TCG takes a combinatorial approach, it outputs a high number of test artifacts. This was particularly evident with the “Registration” page, where there were 17 input fields, which would have resulted in more than 191 million combinations of test scenarios. The computing power available for the experiment was not enough to generate such an amount of data. Thus,

| Table 5 | Total number of test artifacts produced. |
|---------|----------------------------------------|
| Tool name | New Ads | Payment | Registration | Total |
| Spawner  | 100     | 100      | 100          | 300   |
| TCG      | 433     | 108      | –            | 541   |
| Proposed Approach | 27     | 17       | 84           | 128   |

Two other input fields were not covered since checkboxes and radio buttons were not supported by two of the tools.

---

\[\text{Footnote:} \]

\[\text{Four input fields were not covered since checkboxes and radio buttons were not supported by two of the tools.}\]
To be able to compare the output of the tools, we needed to normalize the data. We manually analyzed the generated test artifacts with the goal of extracting input field states that each test case or scenario is testing. For example, suppose that two test cases were generated, one is “User Id is Alice” and the other is “User Id is Bob”. If both of these test cases are targeting “User Id input is valid”, then we consider these two test cases as one input field state. The result of the manual analysis was a list of all the input field states that are covered by all three tools. Each tool output is then compared to the list of states to see how many states each tool covers. The result contains the coverage of each tool within the set of all the covered states of all three tools. Note that the order of the input field states is irrelevant for all three tools.

Table 6 shows the total number of unique input field states that were found for each of the three Web pages we covered. To compute this, we first extracted the input field states produced by each tool. We then combined these to produce the “total” set of input field states. We removed duplicates, which occur if multiple tools produce the same input field state. This, in effect, gave us a list of all covered tests that all three tools came up with.

Table 7 shows the coverage of the input field states for each tool. The “# States” column shows the number of input field states that were covered by a tool for a Web page. The “Coverage” column shows the percentage of states that were covered for that Web page. The “Total Coverage” column shows the percentage of states that were covered for all three Web pages. A tool can achieve a maximum of 100% coverage, if it is able to produce all the input field states that all three tools produced.

As can be seen from the table, our approach had a much higher coverage than the other two tools. We should note that there may be input field states that none of the three tools were able to produce. But, as we are only concerned with comparing these three tools, we focus on the set of input field states produced by the three.

5.3 Comparative Analysis

Table 7 shows that our proposed approach covers a majority of the states, outperforming the two other tools. It is noticeable that the knowledge-based approach covered almost all the states extracted from the Web page under test. This is mainly because of the fact that the database was able to gather exact or dictionary matches to the indicated input fields in the Cyclos page. The missed states were the ones that were not covered by the applications used during database population. That is, these were not part of the current knowledge base. Two examples are “Expiry Date = empty”, which was covered by TCG in the “New Advertisement” page, and “Web Site was set to an IP address”, which was covered by Spawner in the “Registration” page. Both of these cases were not available in the database. The inclusion of more information from past Web applications can solve these misses.

It can be said that even if our approach produced the least number of test artifacts, in the end, it covered more input states than the two other tools.

As mentioned in Sect. 3.4, our method employs a one-node replacement approach. This was essential in limiting the number of generated scenarios produced by our tool. One foreseen advantage is that potential users will not be overwhelmed by the number of output produced by the tool. It is worth noting that, even with a relatively small number of scenarios, our tool still outperformed the two others. This may be attributed to the domain-specific nature of test information contained in our knowledge base.

It is also worth noting that Spawner produced test data closer to being executable compared to our tool. Also, TCG produced combinations of inputs that may be usable and even critical to some software-under-test. In this case, it is worthy to note these particular strengths of the two other tools and look for ways to integrate them in our future work.

6. Conclusion and Future Work

We proposed an approach that generates test scenarios for Web applications by reusing testing information from previous software tests. The test engineer first creates a base scenario, and then uses that to search for related scenarios in the database.

We evaluated our proposed approach from three perspectives. First, we compared the results of manually created test scenarios by professional software test engineers and test scenarios generated by our tool. We found that our tool can produce more scenarios more quickly, including scenarios that the professionals did not create. Second, we surveyed several professional test engineers, who replied that our tool was easy to understand and should be fairly easy to integrate to industry-level testing. Finally, we found that our approach has higher coverage in terms of input states compared to two existing tools.
Future work includes the following. First, we need to consider how we can better build a dictionary, as well as other techniques for determining synonyms. Second, a method to incorporate outputting of actual, testable values and combinatorial scenario node replacement can also be pursued. Finally, we need a way to identify noisy scenarios.

References

[1] D. Xu, H. Li, and C.P. Lam, “Using adaptive agents to automatically generate test scenarios from the UML activity diagrams,” Proc. APSEC ’05, pp.385–392, 2005.
[2] A. Andrews, J. Offut, and R. Alexander, “Testing Web applications by modeling with FSms,” Software & Systems Modeling, vol.4, no.3, pp.326–245, 2005.
[3] S. Fujiwara, K. Munakata, Y. Maeda, A. Katayama, and T. Uehara, “Test data generation for Web application using a UML class diagram with OCL constraints,” J. Innovations in Systems and Software Engineering, vol.7, no.4, pp.275–282, 2011.
[4] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. Ernst, “Finding bugs in Web applications using dynamic test generation and explicit-state model checking,” IEEE Trans. Softw. Eng., vol.36, no.4, pp.474–494, 2010.
[5] W. Halfond and A. Orso, “Improving test case generation for web applications using automated interface discovery,” Proc. ESEC-FSE ’07, pp.145–154, 2007.
[6] C. Pacheco and M. Ernst, “Randoope: Feedback-directed random testing for Java,” Proc. OOPSLA 2007 Companion, pp.815–816, 2007.
[7] F. Gross, G. Fraser, and A. Zeller, “Search-based system testing: High coverage, no false alarms,” Proc. ISSTA 2012, pp.67–77, 2012.
[8] C. Krueger, “Software reuse,” ACM Computing Surveys, vol.24, no.2, pp.131–183, 1992.
[9] A. Mili, R. Mili, and R. Mittermeir, “A survey of software reuse libraries,” Annals of Software Engineering, vol.5, no.1, pp.349–414, 1998.
[10] S. Sim, M. Umarji, S. Ratanotayanon, and C. Lopes, “How well do search engines support code retrieval on the Web?,” ACM Trans. Software Engineering and Methodology, vol.21, no.1, pp.4:1–4:25, 2011.
[11] R. Holmes and R. Walker, “Systematizing pragmatic software reuse,” ACM Trans. Software Engineering and Methodology, vol.21, no.4, pp.20:1–20:44, 2012.
[12] C. Kaner, “Cem Kaner on scenario testing: The power of ‘what-if...’ and nine ways to fuel your imagination,” Software Testing and Quality Engineering Magazine, vol.5, no.5, pp.16–22, 2003.
[13] R. Lacanienta, S. Takada, H. Tanno, and M. Oinuma, “A knowledge-based approach for generating test scenarios for Web applications,” Proc. 25th International Conference on Software Engineering and Knowledge Engineering (SEKE 2013), pp.166–171, 2013.
[14] M. Landhausser and W. Tichy, “Automated test-case generation by cloning,” Proc. 7th International Workshop on Automation of Software Test (AST 2012), pp.83–88, 2012.
[15] W. Janjic and C. Atkinson, “Utilizing software reuse experience for automated test recommendation,” Proc. 8th International Workshop on Automation of Software Test (AST 2013), pp.100–106, 2013.
[16] UMLet, http://www.uml.com [accessed Feb. 20, 2013]
[17] Tapestry5 Hotel Booking, http://tapestry zones.apache.org:8180/tapestry5-hotel-booking/ [accessed Feb. 20, 2013]
[18] OLAT, http://www.olat.org/ [accessed Feb. 20, 2013]
[19] Spawner, http://sourceforge.net/projects/spawner/files/spawner/ [accessed May 30, 2013]
[20] Test Case Generator, http://testcasegenerator.codeplex.com/ [accessed May 30, 2013]
[21] Cyclos Project, https://www.cyclos.org/cyclos_en/do/member/editAd [accessed May 30, 2013]