Test Data Generation for Web Applications: A Constraint and Knowledge-based Approach

Hibiki Saito*, Shingo Takada
Dept. of Information and Computer Science
Keio University
Yokohama, Japan

Haruto Tanno, Morihide Oinuma
Software Innovation Center
NTT Corporation
Tokyo, Japan

Abstract—Software testing is an important part of the software development process. Much work has been done on automating various parts of testing. In previous work, we had proposed a knowledge-based approach to generate test scenarios for Web applications. However, our previous work did not account for generation of actual test data. Thus, in order to execute the test scenarios, the user would need to (manually) create the test data. This paper proposes an approach to generate test data for our previously proposed test scenario generation tool. Our approach can generate two types of test data: constraint-based test data and database-based test data. Our tool can now automatically execute the combined test scenario and test data. We confirmed the usefulness of our approach through a case study.

Keywords—test data generation; Web applications; test scenario

I. INTRODUCTION

Software testing is an important part of the software development process, but it can be time-consuming and very costly. Thus much work has been done on automating various aspects of testing, such as test case generation and test case execution.

In previous work, we had proposed a knowledge-based approach to generating test scenarios for Web applications [1]. A test scenario represents a sequence of steps that are taken to transition from one Web page to another. We generated test scenarios by using test scenario information from previous Web applications that were stored in a knowledge base. Although we had shown that our approach was able to generate many useful test scenarios that professionals were not able to create manually, there was one important drawback. We could generate test scenarios, but we could not generate the test data themselves. A tester would need to manually generate the test data based on the test scenarios to actually conduct the test.

We thus propose an approach to generate test data for our previously proposed test scenario generation tool. The main contributions of this paper are as follows:

1. Extension of our previously proposed tool to generate test data.
3. Evaluation through a case study.

The rest of this paper first starts with a discussion of related work. Section III then describes our approach. Section IV evaluates our approach. Section V makes concluding remarks.

II. RELATED WORK

Much work on test data generation has been done [2]-[9]. Pacheco, et al proposed feedback-directed random testing, and implemented it as Randoop [2]. Basically, test data is generated randomly, but the generation process is guided by the execution of previous inputs.

Another approach is based on symbolic testing, which was originally proposed by King [3]. In symbolic testing, a program is “executed” using symbols rather than concrete values for each variable. These symbols are collected to form expressions which can be solved to obtain concrete values, i.e., test data, each of which will result in the execution of a certain program path. Concolic testing addressed the issue of redundant execution in symbolic testing by combining concrete execution with symbolic execution [4].


Finally, model-based testing generates test data using models and/or formal specifications. Fujiwara, et al [9] proposed a formal approach based on UML class diagrams and OCL constraints. It was shown to be a powerful method, but it requires a tester/modeler to be well-trained in formal modeling.

Work has been done on test data generation for Web applications. But they have issues such as being applicable only to PHP applications [8] or needing information that requires someone that is well trained [9]. Such approaches cannot be used for our test scenario generation tool. Thus, we propose our approach to work on top of our previous test scenario generation tool, which can be used for any Web applications since it does not require a certain language such as PHP, and it does not require the tester to be well trained in a certain technology.

* Currently at Hitachi, Ltd.
III. TEST DATA GENERATION

A. Previous Work

Our test data generation is based on our previous work on test scenario generation. We thus first give an overview of our previous work.

The tester first creates a base scenario of the Web application under test. This base scenario corresponds to the basic normal steps that a user takes for the current Web page. For example, a base scenario for logging in may include steps such as “input user id”, “input password” and “submit information” (Fig. 1 (a)).

Our tool then searches a database containing information on each step (called “scenario node”). The goal of this search is to find nodes that were used in previous tests that can replace nodes in the current base scenario. A dictionary is also employed to account for synonyms, e.g., “user name” and “user id” can be considered as synonyms. Our tool generates related test scenarios, each differing with the base scenario by one node which had been replaced. Thus for example, given the base scenario in Fig. 1 (a), a related scenario (shown in Fig. 1 (b)) may be generated where the first node is replaced.

B. Overview of our proposed approach

Figure 2 shows the overview of our proposed approach. The initial step of constructing a base scenario with our Scenario Editing Tool is the same as our previous work, except the tester will need to specify constraints and some other extra information, specifically Input Type and Concrete Value (subsection III-C), for test data generation and execution.

Our Scenario Analyzer analyzes the base scenario and searches for nodes in the scenario node database to use for replacement. The dictionary is used to check for synonyms. The Test Case Generator generates the related test scenarios (as in our previous work) and also test data. We consider a test case to be a combination of a test scenario and test data.

Our Test Case Generator generates two types of test data. The first type is constraint-based test data, which is generated by encoding constraints for each node. For example, if a node is concerned with inputting numbers within a textfield, then we encode that information within that node. When generating test data, we can then (randomly) generate a string that includes a non-number for that textfield.

The second type is database-based test data. This type of test data is reuse of data that was used in a previous test case. For example, if a node contains information that “johnny” was used in a previous test case, then that can be used.

Finally, our Test Case Executor automatically executes the set of ranked test cases. Test cases are ranked based on frequency, i.e., if nodes have been used often in previous tests, then those test cases have a higher ranking.

C. Scenario Node

Table I shows information that is contained within the scenario node database. The new types of information that have been added are “Input Type” and “Concrete Value”. “Input Type” was added to prevent invalid scenarios from being generated. For example, in our previous tool, if the Process Name and Node Type matched, a textfield node may be replaced with a radio button node. This obviously would not lead to an executable test case. “Concrete Value” stores values that were used in previous tests, and is used when generating database-based test data.

![Diagram of the test data generation process](image)

Figure 1. Base scenario and related scenario

![Overview of proposed approach](image)

Figure 2. Overview of proposed approach

<table>
<thead>
<tr>
<th>Description</th>
<th>Table I. NODE INFORMATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>ID for node.</td>
</tr>
<tr>
<td>Process Name</td>
<td>The type of processing done by the Web page. (e.g., login, registration)</td>
</tr>
<tr>
<td>Node Type</td>
<td>The type of node. (e.g., PROCESSING, INPUT)</td>
</tr>
<tr>
<td>Input Type</td>
<td>The type of input. (e.g., textfield, radio button)</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Name of attribute. (e.g., username, password)</td>
</tr>
<tr>
<td>Value Class</td>
<td>The type of value. (e.g., empty, exist)</td>
</tr>
<tr>
<td>Concrete Value</td>
<td>Values that were used in previous tests.</td>
</tr>
<tr>
<td>Frequency</td>
<td>The number of times this node was used in previous tests.</td>
</tr>
</tbody>
</table>
When a tester specifies a base scenario, he/she may describe constraints for each node. Table II shows text input constraints that the tester can encode for nodes in the base scenario. This did not exist in our previous work, and enables the generation of constraint-based test data. Note that this information is not saved in the database.

For example, the constraint “more” specifies that the length of a string must be greater than a certain number. Thus, in case of a password, the tester can specify “more=7”, which means that the length of the password must be greater than or equal to 7. The constraint “must” specifies that the string must include a certain type of character. Thus, in case of a password, the tester can specify “must=NUMBER AND ALPHABET AND SYMBOL”, which means that the password must have at least one number, one alphabetical letter, and one symbol. If the tester wants to specify a specific character, then he/she can specify “must=‘AbC’”, which means that the string “AbC” must be included.

These constraints could have been specified using existing constraint languages, such as OCL, but we intentionally kept this simple so that testers can easily specify the constraints without special training.

### D. Test Data Generation

We generate test data by combining the base scenario with information in the knowledge base. Our previous work generated related scenarios by replacing nodes [1]. In this paper, we go one step further; we automatically generate test data based on two approaches: (1) database-based approach and (2) constraint-based approach.

In the database-based approach, there are three ways to generate data:

1. **Manual Input:** The tester manually specifies the test data.
2. **Altered Data:** The value in the original node of the base scenario is altered. In the case of text input, one character in the value (string) is randomly changed.
3. **Previous Data:** We check the Concrete Value field of the node that we will use to replace. If a value exists, then it is used. Of course, if there are no values in the Concrete Value field, then our tool cannot automatically generate test data.

In the constraint-based approach, we can generate data as follows:

1. **Manual Input:** The tester manually specifies the test data.
2. **Automatic Generation:** Randomly generate data each of which does not satisfy one constraint, but satisfies all other constraints.

Based on the above two test data generation approaches, our tool operates as follows. Our tool searches for nodes in the node database that match the Process Name and Attribute Name, i.e., this search corresponds to the search for nodes that can be used to replace nodes in the base scenario, and generate related scenarios.

Next, a data selection window pops up (Fig. 3). In this example, the Process Name is “LOGIN”, Attribute Name is “USER ID”, the Value Class is “DOES NOT EXIST”, and the data in the original node is “johnny825”. The data selection window shows four options: (1) skip generation of test data, (2) manually specify test data, (3) use test data that has been automatically generated by altering the data in the original node (which results in “johuny825”), and (4) use data that exists in the Concrete Value field (which results in “johnnyjohnny”). The tester chooses one of these options. This process of data selection will continue until all nodes that can be replaced have been processed.

![Figure 3. Concrete value selection menu for database-based approach](image)

2. **Automatic Generation:** Randomly generate data each of which does not satisfy one constraint, but satisfies all other constraints.

### TABLE II. CONSTRAINTS FOR TEXT INPUT

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>More</td>
<td>String length must be greater than specified number</td>
</tr>
<tr>
<td>Less</td>
<td>String length must be less than specified number</td>
</tr>
<tr>
<td>Only</td>
<td>String can only take a certain type of character</td>
</tr>
<tr>
<td>Must</td>
<td>String must include a certain type of character</td>
</tr>
<tr>
<td>Must Not</td>
<td>String must not include a certain type of character</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraint</th>
<th>VALUECLASS</th>
</tr>
</thead>
<tbody>
<tr>
<td>more=XX</td>
<td>SHORT</td>
</tr>
<tr>
<td>less=XX</td>
<td>LONG</td>
</tr>
<tr>
<td>only=XX AND YY</td>
<td>NOT ONLY XX AND YY</td>
</tr>
<tr>
<td>must=XX AND YY</td>
<td>DOES NOT HAVE XX, DOES NOT HAVE YY</td>
</tr>
<tr>
<td>mustnot=XX AND YY</td>
<td>HAVE XX, HAVE YY</td>
</tr>
</tbody>
</table>
Next, a data selection window pops up (Fig. 4). In this example, the Process Name is “REGISTRATION” and the Attribute Name is “PASSWORD”. The original node had “must=NUMBER” as a constraint; thus the newly generated node will have VALUECLASS as “DOES NOT HAVE NUMBER”. The data selection window shows three options: (1) skip generation of test data, (2) manually specify test data, and (3) automatic generation of test data, which in this case is “pH@FRY”.

The automatic generation of constraint-based test data is done as follows:

1. Decide on string length: If there is a “more” or “less” constraint, then the length is set to that value. If both “more” and “less” constraints exist, then the length is set to the median value. If no constraint exists, then the length is set to the original string length.
2. Decide on the characters to be used: The decision is made based on the “only”, “must”, and “mustnot” constraints.
3. Generate string: Based on the above, a string is randomly generated.
4. Confirm: If the generated string does not satisfy a string constraint, then go back to step 3. This last step is done as there may be multiple constraints for a node, and we need to check if the randomly generated string will not violate the other constraints.

**E. Test Case Execution**

The generated test data is executed with our Test Case Executer. Our Test Case Executer uses Selenium WebDriver [10], which enables the automatic operation of a Web page. For each test data, the following four steps are executed:

1. Launch a Web browser.
2. Execute the operation specified in each node (e.g., text input, button press).
3. Save a screenshot of the result.

A screenshot of the result is saved in step 3 so that the tester can check the result as necessary.

IV. CASE STUDY

We conducted a case study to evaluate our approach focusing on the following three research questions:

- **RQ1**: When different users use our tool to generate test data for the same Web application, how different are they?
- **RQ2**: How different are the tool-generated test data and test data that are manually created by a professional?
- **RQ3**: For test data that were created manually, can they be used to update the knowledge base?

**RQ1** is posed to check how the tool can be used. Since the test data is generated with a knowledge base, if all users use the same knowledge base, then there should not be much of a difference between different users.

**RQ2** checks what types of test data cannot be generated. But since this is likely tied to how well the knowledge base is made, **RQ3** is a follow up to **RQ2**, i.e., if the knowledge base had been made better, what would the difference between the tool-generated test data and manually created test data be?

**A. RQ1: Difference between Different Tool Users**

Six students majoring in computer science took part in our case study. After first getting instructions on how to use our tool, each student “played” with the tool. They were then instructed to generate test data for the login, registration and search Web pages for three Web applications: Cyclos [11], Tapestry5 [12], and Redmine [13].

Table IV shows the results for the total number of generated test data and the total amount of time that each subject took. The result shows that overall there was not much of a difference between the subjects. The time taken to draw the base scenario and then generate the test cases was between 73 and 86 minutes for a difference of 13 minutes (15%). The total number of generated test data ranged between 118 and 132 for a difference of 14 (11%).

**TABLE IV. TOOL GENERATED TEST DATA RESULTS**

<table>
<thead>
<tr>
<th>Subject</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total #</td>
<td>118</td>
<td>123</td>
<td>123</td>
<td>129</td>
<td>120</td>
<td>132</td>
</tr>
<tr>
<td>Time (min)</td>
<td>76</td>
<td>86</td>
<td>81</td>
<td>80</td>
<td>78</td>
<td>73</td>
</tr>
</tbody>
</table>

A closer inspection of the generated test data found that the total number of types of test cases, i.e., scenarios, was 134. Among these, all six subjects generated the same 106 types (79%), while five out of the six subjects generated 121 types (90%). The actual test data itself was not always the same, because different concrete values from past test data were chosen for the same scenario, and the constraint-based test data were different because they were randomly generated. This clearly indicates that regardless of the tester, most of the generated test cases will be the same.

We also checked for invalid test cases, and found three. These occurred when the user erroneously manually created test data. For example, the VALUECLASS stated “HAVE NUMBER", but the test data was “ABCD", which clearly does not have a number included. All cases where the user decided to automatically generate the test data were valid.
B. RQ2: Difference between Tool-Generated Test Data and Manually Created Test Data

We asked a professional software engineer to manually create test cases for the three Web applications. The software engineer took a total of about 175 minutes to create the test cases, which is more than twice as long as it took the students to generate the test cases. The total number of test cases that the professional engineer created was 114. This is less than the number of test cases that each of the subject created. Thus we can say that our tool can be used by testers that do not necessarily have deep knowledge of how to create tests.

We next check the type of test cases that were generated. Fig. 5 (a) shows the breakdown in Venn diagram format.

![Venn diagram](image)

Figure 5. Generated test cases before and after knowledge base update

In Fig. 5 (a), “Pro” shows the number of test cases that the professional engineer manually created, while “Tool” shows the number of test cases that the students generated with our tool. Note that for the tool generated test cases, we did not use the entire set; instead, we used the 121 test cases that five out of the six subjects generated, since we believe that this will better represent the tool generated set of test cases.

From Fig. 5 (a), 69 test cases were created only by the professional engineer, while 76 test cases were generated only by our tool. There was an overlap of 45 test cases that both the professional and our tool created. In other words, 39% (=45/(69+45)) of the test cases created by the professional were also generated by our tool. This is not necessarily a high number. But since our tool is knowledge-based, this is likely due to not enough information being stored in our knowledge base. We consider this further in the next subsection.

C. RQ3: Updating the Knowledge Base

Since our tool is knowledge-based, the results in the previous subsection may have been better if the professional base was populated better. We thus added the knowledge base with the information from the test cases that the students had made. We then automatically generated the test cases again, using the base scenarios that the student subjects had made. Fig. 5 (b) shows the results.

As Fig. 5 (b) shows, by populating the knowledge base, 94% (=107/(7+107)) of the test cases that the professional created can be generated by our tool. The remaining seven can be categorized into the following two types: (1) No concrete value possible due to constraints, and (2) impossible to add to database due to ambiguity.

Concerning the first type, when our tool generates database-based test data, the test data must satisfy the constraints that have been set for that node. Test data that do not satisfy constraints can only be generated through constraint-based generation. For example, the professional engineer specified a test case where the input text is one letter long. In our tool, this would correspond to a test case where VALUECLASS=ONE LETTER. Unfortunately, if the node has a constraint which states that the input text must be three or more letters long, it will be impossible to generate test data, since the length of a text cannot be both one letter and more than three letters long.

As for the second type, in some cases, the professional created a test case where the meaning will depend on the context as well as constraints. In one case, the professional specified “invalid characters for user name”. An invalid character can differ between applications, and thus can only be handled through constraint-based test data generation.

V. CONCLUSION

We proposed a test data generation scheme based on database and constraints. Along with our previous work on test scenario generation, we can automatically generate test cases, i.e., combination of test scenarios and test data. Our tool can automatically execute these test cases. We showed the results of a case study.

The main part of future work concerns the knowledge base itself. As with any knowledge-based approaches, our tool is completely reliant on what is currently included in the knowledge base. We thus need to consider how we can systematically update the knowledge base. We also need to consider scalability issues; updating the database will lead to more test cases, but will there be any tendencies as to what type of test cases are more useful than others?

REFERENCES