POGen: A Test Code Generator Based on Template Variable Coverage in Gray-Box Integration Testing for Web Applications

Kazunori Sakamoto[1], Kaizu Tomohiro[2], Daigo Hamura[2], Hironori Washizaki[1], and Yoshiaki Fukazawa[1]

[1] Waseda University
Tokyo, Japan
kazuu@ruri.waseda.jp, washizaki@waseda.jp, fukazawa@waseda.jp,
[2] Google Japan Inc.
Tokyo, Japan
tkaizu@google.com, daigoh@google.com

Abstract. Web applications are complex; they consist of many subsystems and run on various browsers and platforms. This makes it difficult to conduct adequate integration testing to detect faults in the connections between subsystems or in the specific environments. Therefore, establishing an efficient integration testing method with the proper test adequacy criteria and tools is an important issue.
In this paper, we propose a new test coverage called template variable coverage. We also propose a novel technique for generating skeleton test code that includes accessor methods and improves the template variable coverage criterion, using a tool that we developed called POGen. Our experiments show that template variable coverage correlates highly with the capability to detect faults, and that POGen can reduce testing costs.

Keywords: software testing, web application, test coverage, test code generation, template engine

1 Introduction

The importance of web applications has grown immensely with the popularization of the Internet. Web applications are based on the client-server model, and require collaboration among client and server programs. This indicates that web applications consist of various subsystems such as web servers, authentication servers and database servers. Moreover, client programs run on various browsers and platforms. Some faults might occur in the connections between the subsystems or in the specific environments [1]. Therefore, an efficient and comprehensive testing method is required to eliminate these faults.

Software testing methods can be roughly classified by their level and technique. There are four levels of testing during the development process: unit testing, which verifies the functionality of a specific section of the code; integration testing, which verifies the interfaces between components; system testing,

which verifies that the entire software meets its requirements; and acceptance testing, which verifies that the requirements are met from the user's perspective. For simplicity, we consider any testing except unit testing as integration testing in this paper. Testing methods can also be classified into three techniques: white-box testing (a.k.a. structure based testing), which utilizes knowledge of how the code is implemented; black-box testing (a.k.a. specification based testing), which tests the functionality of the code without any knowledge of how the code is implemented; and gray-box testing, which utilizes partial knowledge of the internal data structures and algorithms.

Testing frameworks such as JUnit automate the testing process by writing test code. Test code constitutes an executable program including test cases. A test case consists of a test scenario, which invokes functions of the production code, and a test oracle, which determines whether the program works as expected by observing the program state as the test scenario is executed.

Template engines are typically introduced during web application development as part of the web framework, such as Struts and Ruby on Rails. There are two types of template engines: those that run on the server side, such as ejs [2] and JSF [3], and those that run on the client side, such as Closure Templates [4]. A template engine generates an HTML document by embedding string representations of variables or expressions referring to the states of executable programs into the HTML template. Therefore, web applications must be able to handle both the static content of the HTML document and the dynamic content that changes as the program is executed. Finding faults related to the static content is relatively easy, whereas finding faults related to the dynamic content is difficult because whether the faults are exposed or not depends on the execution path.

In this paper, we propose a new test coverage criterion called template variable coverage. Template variable coverage focuses on the variables and expressions for embedding in HTML templates (hereafter referred to as template variables), which are important for testing a web application's functionality related to the dynamic content of an HTML document. To improve the template variable coverage criterion, we also propose a gray-box integration testing method using a tool that we developed called POGen. POGen generates skeleton test code with accessor methods for template variables by analyzing HTML templates.

The contributions of this paper are as follows:

- Template variable coverage, a new criterion of test adequacy to verify the dynamic content corresponding to the execution results of web applications,
- POGen, a novel test code generator for improving template variable coverage criterion,
- An evaluation of the correlation between template variable coverage and the capability to detect faults and
- An evaluation of POGen with respect to its ability to reduce writing and maintenance costs.

POGen is released as open source software on `http://code.google.com/p/pageobjectgenerator/`.

2 Motivating example

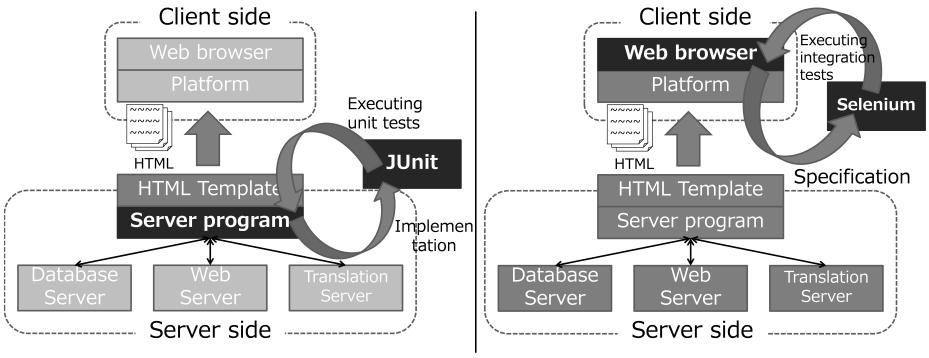

Fig. 1. White-box unit testing (left) and black-box integration testing (right) for a sample web application

List 1.1. An HTML template of the input page using Closure Templates

```
{template .addition}
  <form action="result">
    <input type="text" name="left"> + <input type="text" name="right">
    <input type="submit" name="calc" value="Calculate">
  </form>
{/template}
```

List 1.2. An HTML template of the result page using Closure Templates

```
{template .result}
  <p>{$left}+{$right}=<div>{$answer}</div></p>
  <ul>{call .items}{param list: $histories /}{/call}</ul>
{/template}
{template .items}
  {foreach $item in $list} <li>{$item}</li> {/foreach}
{/template}
```

In this section, we present an example to demonstrate our technique. We shall consider a web application for a simple calculator supporting addition. The calculator has two additional features for translating string representations of numbers such as "fourteen" and for recording and showing past inputs. Figure 1 shows the architecture of the web application. The web application consists of three servers: a web server for handling web pages with a template engine, a translation server for handling number strings and a database server for managing the input history.

The web application consists of two web pages: an input page for inputting numbers to add and a result page for showing the result of the calculation. Lists 1.1 and 1.2 show the HTML templates of the input and result pages using Closure Templates. The result page shows the translated numbers, the addition result and the history of the given numbers and results. We discuss three testing methods regarding this web application.

2.1 White-box unit testing

White-box testing is usually used for the unit testing of web applications and not for integration testing because it is difficult to consider implementations of all subsystems and the relations between these implementations due to extremely large subsystems. White-box unit testing is less dependent on the tester than black-box testing because it is based on the implementation, including test coverage. Although white-box unit testing exhaustively verifies features, it concentrates on only one module at a time, so the connections between the modules are not tested. For example, white-box unit testing cannot find mismatches between variable names in the HTML templates and in the server program, or the invocation of an inappropriate function from the server program. Therefore, faults that can be found with white-box unit testing are limited.

2.2 Black-box integration testing

Black-box testing is usually used for the integration testing of web applications as opposed to white-box testing. Black-box testing only requires the testers to know what the web application is supposed to do, and the testers can easily try to manipulate the web application on a web browser. Black-box integration testing can be done at a low cost, and can find faults between connections of modules on the real environment. However, it is hard to exhaustively verify the dynamic content of the HTML document because the testers have no information on how the application is implemented. Black-box integration testing also depends heavily on the tester. For example, a tester of our sample web application may only concentrate on the answer value and not test the left and right values to verify the calculation feature. As another example, a tester may leave out the translation feature. Therefore, black-box integration testing may overlook faults due to its dependency on the tester and the lack of test adequacy criteria.

2.3 Gray-box integration testing

To find faults that can be overlooked by both white-box unit testing and black-box integration testing, we propose a gray-box integration testing method for verifying web applications. This method makes use of information gathered from the HTML templates and the application's specifications.

List 1.2 contains four template variables: {`$left`}, {`$right`}, {`$answer`} and {`$item`}. Note that `$histories` and `$list` are not template variables because they are not inserted into the HTML document replacing them with their values during execution. The template variables indicate the dynamic content of the HTML document corresponding to the outputs from the server program of the web application, and they can be used to verify the connections between subsystems. For example, faults in the addition feature are exposed through {`$answer`}, while those in the translation feature are exposed through {`$left`}, {`$right`} and {`$answer`}, and those in the history feature are exposed through {`$item`}. Therefore, gray-box integration testing allows testers to find faults efficiently and

exhaustively by exposing the dynamic content of the HTML document marked by template variables.

3 Template variable coverage

Template engines replace template variables with the string representations of the template variables' values. We can extract the dynamic content of an HTML document by analyzing HTML templates; template variables are marked by special notations, such as {`$answer`} in List 1.2. Note that we make no distinction between variables and expressions in template variables.

We propose a new coverage criterion called template variable coverage which indicates test adequacy with respect to the dynamic content of the HTML document. Template variable coverage allows testers to conduct exhaustive gray-box integration testing. Moreover, it provides a quantitative measure for assessing the quality of a set of test cases.

Definition 1. Let C_{tmp} be the template variable coverage, V_{all} the number of all template variables, and V_{test} the number of template variables which are referred by test cases during testing. C_{tmp} is defined by formula (1). Note that $V_{all} \supseteq V_{test}$ and $0 \leq C_{tmp} \leq 1$ hold.

$$C_{tmp} = \frac{V_{test}}{V_{all}} \tag{1}$$

List 1.3. A test case with JUnit and Selenium for verifying the addition feature

```
1 WebDriver driver = new ChromeDriver();
2 // some code to initialize the driver
3 driver.findElementByName("left").sendKeys("1");
4 driver.findElementByName("right").sendKeys("two");
5 driver.findElementByName("calc").submit();
6 assertEquals("3", findElement(By.cssSelector("p > div")).getText());
```

List 1.3 shows a conventional test case to verify the addition feature of the web application. This test case only refers to one template variable, {`$answer`}, out of the four template variables: {`$left`}, {`$right`}, {`$answer`} and {`$item`}. Thus, the template variable coverage for this test case is 25%. It is 75% if a test case refers to {`$left`}, {`$right`} and {`$answer`}.

Assumption 1. Template variable coverage is a test adequacy criterion that indicates how exhaustively the test cases observe outputs from web applications. We therefore assume that template variable coverage correlates positively with the test quality, or the capability to find faults with the set of test cases.

4 POGen: test code generator to improve the coverage

We propose a novel technique to generate skeleton test code for improving the template variable coverage criterion using POGen. POGen supports gray-box integration testing based not only on the specifications of the web application but also on the HTML templates.

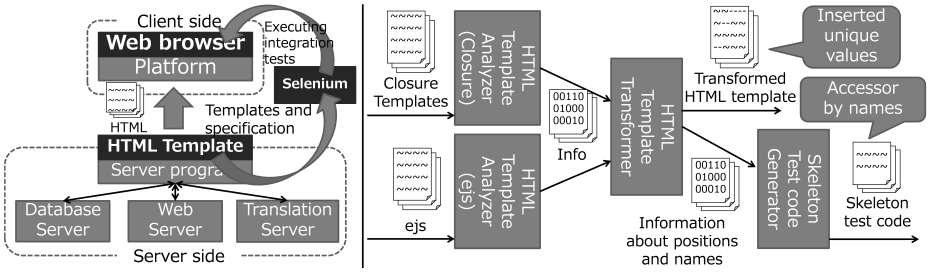

Fig. 2. Gray-box integration testing for the sample web application (left) and the architecture of POGen (right)

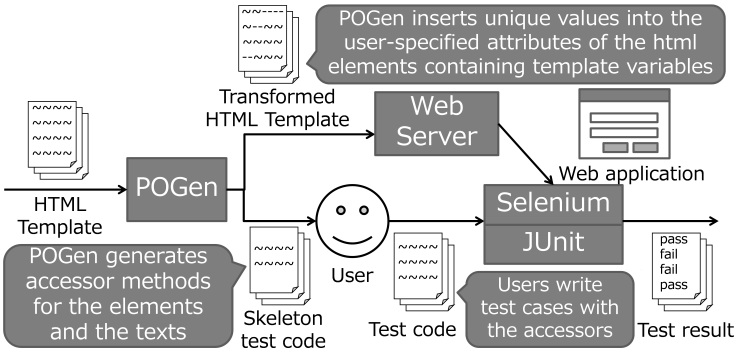

Fig. 3. Illustration of the role of POGen in gray-box integration testing

The left side of Figure 2 shows the relation between the sample web application and gray-box integration testing based on the HTML template. To help conduct gray-box integration testing, we developed a tool called POGen for analyzing HTML templates and generating skeleton test code with Selenium and JUnit. POGen extracts template variables which indicates the dynamic content and the operable HTML elements such as `<a>`, `<link>`, `<input>`, `<textarea>`, `<select>` and `<button>` elements. POGen also generates skeleton test code designed with the PageObject design pattern [5], which has high maintainability and contains accessor methods for the template variables and the operable HTML elements. POGen thus reduces the writing and maintenance costs.

The right side of Figure 2 shows the architecture of POGen. POGen consists of three components: the HTML template analyzer, the HTML template transformer and the test code generator. POGen generates accessor methods for HTML elements containing template variables and operable HTML elements. To increase maintainability, the methods depend not on the HTML structure but on the names of template variables and operable HTML elements. POGen transforms HTML templates for exposing HTML elements containing template variables and operable HTML elements for test purposes only.

Figure 3 shows the process of gray-box integration testing with POGen.

1. Testers feed the HTML template files to be tested into POGen.
2. POGen analyzes and transforms the HTML templates by inserting unique values into the specified attributes of the HTML elements containing template variables and the operable HTML elements.
3. POGen generates skeleton test code with accessor methods for the HTML elements by referring to the inserted attribute values.
4. Testers enhance the generated skeleton test code and write test cases.
5. The transformed HTML templates are deployed on the web server.
6. Gray-box integration testing is conducted by running the test cases.

4.1 HTML template analyzer

The POGen HTML template analyzer extracts the HTML elements by finding, naming and then analyzing the template variables and operable HTML elements. After these three steps, which are described below, the HTML template analyzer passes the positions of the template variables and the operable HTML elements into the HTML template transformer.

The HTML template analyzer first finds the template variables and the operable HTML elements by parsing the HTML template, and records positions. For the HTML template in List 1.1, the analyzer will find three operable elements of `<input>`. For the HTML template in List 1.2, the analyzer will find four template variables: {`$left`}, {`$right`}, {`$answer`}, and {`$item`}.

In the next step, the analyzer determines the names of the extracted template variables and the extracted operable HTML elements to generate accessor methods. The analyzer names the template variables with the texts of the template variables removing the head and tail sign characters and replacing the other sign characters with underscore characters '_'. The analyzer also names the operable HTML by concatenating the types of the HTML elements with the `id` attribute values, `name` attribute values or the texts. For the HTML template in List 1.1, the analyzer will name the first operable elements `"INPUT_left"`. For the HTML template in List 1.2, the analyzer will name {`$left`} `"left"`.

In the analyzing step, the analyzer determines if the extracted HTML elements appear in loop statements, such as `for` and `foreach` statements. Depending on whether the template variables are repeated or not, POGen generates accessor methods returning a list of HTML elements or a single HTML element, respectively. In List 1.2, {`$item`} is a repeated template variable.

POGen currently supports Closure Templates, ejs, erb and JSF. We can easily extend POGen by adding parsers for other template engines.

4.2 HTML template transformer

The HTML template transformer inserts unique values for user-specified attributes, such as `id` and `class`, into HTML elements containing template variables. This allows the generated accessor methods to depend on the names determined by the analyzer and not on the structure of the HTML document.

If the targeted HTML element already contains a value for the specified attribute, then the transformer uses the existing value, either as is for attributes like `id` that only accept a single value, or by inserting another unique value into the existing value for attributes like `class` that accept space-separated values. Users can choose any attribute to be inserted by POGen to avoid changing a web application's behavior. In addition, the transformer inserts special HTML comments into the HTML template so that accessor methods can acquire text representations of the template variables.

List 1.4. An HTML template transformed by the POGen to generate accessor methods

```
{template .addition}
  <!-- POGen,left,{$left} --><!-- POGen,right,{$right} -->
  <p class="_pogen_1">{$left}+{$right}=
    <!-- POGen,answer,{$answer} --><div class="_pogen_2">{$answer}</div></p>
  <ul>{call .items}{param list: $histories /}{/call}</ul>
{/template}
{template .items}
  {foreach $item in $list}
    <!-- POGen:item:{$item} --><li class="item _pogen_3">{$item}</li>
  {/foreach}
{/template}
```

The HTML template in List 1.2 with values of `class` attributes and HTML comments inserted by the transformer is shown in List 1.4. POGen backs up the original HTML templates before they are transformed.

4.3 Test code generator

The test code generator generates skeleton test code containing accessor methods for the extracted HTML elements. The test code is designed by the PageObjects design pattern with JUnit and Selenium. The names of the accessor methods consist of the texts, the types of the HTML elements and the attribute values. The accessor methods can be invoked using only the names determined by the analyzer. This makes the test code independent of the HTML structure.

The PageObjects design pattern modularizes test code in terms of page classes, allowing testers to write test cases as if writing in a natural language by treating page classes and their methods. The page class contains methods corresponding to features provided to the user by the web page, such as login and addition features, as well as fields indicating HTML elements and accessor methods for acquiring information on the page. The modularization improves the maintainability of test code by reducing the amount of necessary code modifications resulting from frequent updates to web applications. Web application updates commonly result in changes to the structure of HTML documents, which necessitates modifications in the operations of DOM trees in the methods of page classes. However, changes in web page features and information are rarely required. Therefore, modifications to test cases are rarely required.

For each web page, POGen generates page classes that have two accessor methods for each HTML element containing template variables. One accessor method returns the object indicating the HTML element. This method allows

testers to write various operations for the HTML element by providing methods for simulating user manipulations from Selenium. The other accessor method returns the string representation of the template variable. POGen also generates an accessor method for each operable HTML element because the operable HTML elements are frequently referred in test code.

POGen requires users to enhance the generated skeleton test code by writing feature methods such as `login` and `add` methods. Users can write test cases after enhancing the page classes. The generated test code is distinguished from the user-written code by the comments `GENERATED CODE START` and `GENERATED CODE END`. When updating the test code to support changes in the web applications, only the generated test code is changed by POGen.

List 1.5. Skeleton test code generated by POGen for the result page template given in List 1.2 for the sample web application

```
1  public class ResultPage extends AbstractPage {
2    /* -------------------- GENERATED CODE START -------------------- */
3    @FindBy(className  = "_pogen_1") private WebElement _left, _right;
4    @FindBy(className  = "_pogen_2") private WebElement _answer;
5    @FindBy(className  = "_pogen_3") private WebElement _item;
6    public WebElement getElementOfLeft () {/*abbrev.*/}
7    public WebElement getElementOfRight () {/*abbrev.*/}
8    public WebElement getElementOfAnswer () {/*abbrev.*/}
9    public List<WebElement> getElementsOfItem () {/*abbrev.*/}
10   public String getTextOfLeft () {/*abbrev.*/}
11   public String getTextOfRight () {/*abbrev.*/}
12   public String getTextOfAnswer () {/*abbrev.*/}
13   public List<String> getTextsOfItem() {/*abbrev.*/}
14   /* ---------------------- GENERATED CODE END ---------------------- */
15 }
```

List 1.5 shows the skeleton test code generated by POGen for the HTML template in List 1.4. The `getElementOfLeft` method returns the `WebElement` object indicating the `<p>` element containing {`$left`} by using the `class` attribute value `_pogen_1`. The `WebElement` object provides various operations such as `sendKeys`, which simulates keyboard inputs, and `click`, which simulates mouse clicks. The `getTextOfLeft` method returns the `String` object of {`$left`} by parsing the HTML comment (e.g. `<!-- POGen,left,{$left} -->`) inserted by POGen. The generated accessor methods are surrounded by `GENERATED CODE START` and `GENERATED CODE END` so that these methods will be updated according to the changes made in the web applications.

List 1.6. Enhanced skeleton test code generated by POGen for the input page template given in List 1.1 for the sample web application

```
1  public class InputPage {
2    /* -------------------- GENERATED CODE START -------------------- */
3    // Abbreviate generated skeleton test code
4    /* ---------------------- GENERATED CODE END ---------------------- */
5    public ResultPage add(String left, String right) {
6      getElementOfINPUT_left().sendKeys(left);
7      getElementOfINPUT_right().sendKeys(right);
8      getElementOfINPUT_calc().submit();
9      return new ResultPage(driver);
10   }
11 }
```

List 1.7. A JUnit test case using the skeleton test code generated by POGen

```
1 public class ResultPageTest {
2   @Test public void add1And2() {
3     WebDriver driver = new ChromeDriver();
4     // some code to initialize the driver
5     ResultPage resultPage = new InputPage(driver).add(1, "two");
6     assertEquals(resultPage.getTextOfAnswer (), "3");
7   }
8 }
```

List 1.6 shows sample test code enhanced by testers to write test cases. List 1.7 shows a test case based on the generated test code in Lists 1.5 and 1.6. This `add1And2` test case asks the web application to add "1" and "two". Then it determines whether the text representation of the template variable {`$answer`} equals the expected value of three. In summary, POGen reduces the writing and maintenance costs of test code by introducing the PageObjects design pattern, and by generating skeleton test code that contains accessor methods.

5 Evaluation

To assess the effectiveness of template variable coverage and POGen, we conducted a set of experiments and compared the results against conventional methods. Specifically, we investigated the following research questions:

- RQ1: Is template variable coverage correlated with a test's capability to detect faults?
- RQ2: How can our approach improve a test's capability to find faults?
- RQ3: How can our approach facilitate writing test code?
- RQ4: How can our approach facilitate maintaining test code?

5.1 Experiment 1

Table 1. The subject web application for Experiment 1

Name	Pages	Test cases	LLOC of production code	LLOC of test code	Killed mutants	All mutants
booking	11	16	2282	634	58	248

To verify Assumption 1 and investigate RQ1, we measured template variable coverage and the number of killed mutants generated by SimpleJester for the subject web applications. Note that the mutation testing tools such as SimpleJester embed faults called mutants, and then measure the detected faults called killed mutants by executing the test with targeted test code. The subject web application was the Seam Framework example called booking (`https://github.com/seam/examples`). Table 1 shows its name, the number of pages

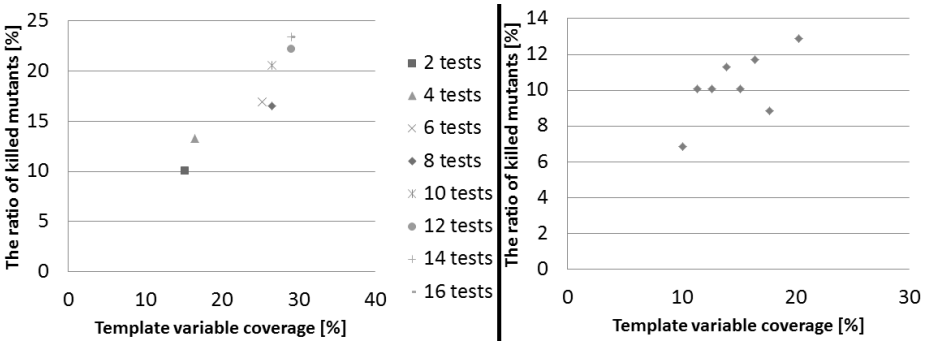

Fig. 4. The graphs which illustrates the correlation with the template variable coverage and the killed mutants for the subject web application with the different number of the test cases (left) and with the same number of the two test cases (right)

and test cases, the logical lines of code (LLOC) of production code and test code, and the number of killed mutants and all of the mutants. To measure the correlation with the template variable coverage and the number of the killed mutants, we randomly reduced test cases.

Figure 1 shows two graphs, whose vertical axis represents the ratio of the killed mutants over all of the mutants, whose horizontal axis represents the template variable coverage and whose labels represent the number of the remaining test cases. Whereas the graph on the left side shows the correlation with the different number of the test cases, the graph on the right side shows the correlation with the same number of the two test cases. As the figure shows, the template coverage correlates highly with the killed mutants, or the capability to detect faults independently of the number of test cases. Therefore, we confirm *Assumption 1* is approved in this example.

5.2 Experiment 2

To investigate RQ2 and RQ3, we conducted an empirical experiment on an open source web application (`https://github.com/TakenokoChocoHolic/almond-choco`), such as that found on TopCoder, which provides online compiling and execution of source code to solve problems. Users can create, edit, delete and solve problems on the web application. A problem consists of a title, a description, an input and an expected result. The web application determines whether the submitted source code is correct by comparing the result from compiling and executing the source code with the expected result. The web application provides five pages: a page where users can see a list of the problems (index page), a page where users can make a new problem (create page), a page where users can edit an existing problem (edit page), a page where users can submit their source code to solve a problem (solve page), and a page where users can compare results (result page).

We measured the time to write two sets of test cases without test oracles and counted the template variables referred by test cases which were enhanced with

test oracles within 20 minutes. We tested three bachelor's and three master's degree students studying computer science (S1, S2 ⋯ and S6).

Table 2. The results of times to write test cases except for test oracles and the number of template variables referred in test cases which are enhanced with test oracles within 20 minutes

A set of test cases	S1	S2	S3	S4	S5	S6	Average
Test cases 1	13 mins	10 mins	14 mins	-	-	-	12.3 mins
with POGen	9 vars	8 vars	7 vars	-	-	-	8 vars
Test cases 1	-	-	-	29 mins	52 mins	34 mins	38.3 mins
without POGen	-	-	-	4 vars	3 vars	5 vars	4 vars
Test cases 2	-	-	-	7 mins	13 mins	15 mins	8.3 mins
with POGen	-	-	-	5 vars	6 vars	8 vars	6.3 vars
Test cases 2	14 mins	36 mins	59 mins	-	-	-	36.3 mins
without POGen	4 vars	4 vars	3 vars	-	-	-	3.6 vars

This experiment consists of two steps: writing test cases without test oracles and writing test oracles to enhance written test cases within 20 minutes with or without POGen. In the first step, the examinees wrote two sets of test cases with the test specification written in natural language. The test cases 1 represent a set of three test cases: updating a problem, solving a problem correctly and wrongly with Python. The test cases 2 also represent a set of three test cases: creating a problem, deleting a problem and solving a problem correctly with Ruby. S1, S2 and S3 wrote the test cases 1 with POGen and then wrote the test cases 2 without POGen. On the other hand, S4, S5 and S6 wrote test cases 1 without POGen and then wrote the test cases 2 with POGen. In the next step, they wrote test oracles for each own test cases within 20 minutes in the same flow.

Table 2 shows the result of this experiment. As the table shows, POGen reduced the writing time of the test cases by approximately 66%. Moreover, the template variables referred by the enhanced test cases with POGen are more than ones without POGen. POGen helps testers to write test code which observes more template variables to detect more faults. Therefore, POGen can reduce costs of writing test code and improve the template variable coverages.

5.3 Experiment 3

Table 3 shows LLOC values for the whole POGen-generated skeleton test code, for the actually used section of the generated skeleton test code and for a manually written test code for the web application in Experiment 2. There are six LLOC values for page classes and test cases using page classes and the sum in Table 3. The test code contains six test cases for creating problems, editing problems, submitting a source code with Python and Ruby correctly and submitting a source code with Python wrongly.

Table 3. The LLOC values of skeleton test code generated by POGen, actually used test code in manually written test code and manually written test code for the web application in Experiment 2

	Index	Create	Edit	Solve	Result	Test cases	Sum
Actually used	82	47	88	44	43	0	304
(Generated by POGen)	(164)	(54)	(164)	(65)	(49)	(0)	(496)
Manually written	12	7	13	6	0	105	143

As the table shows, POGen reduced the writing cost of test cases by generating page classes with accessor methods for HTML elements containing template variables and for operable HTML elements. Through POGen, the testers were able to access HTML elements using only the names of the template variables or the operable HTML elements, without any knowledge of the XPath or CSS selector. POGen successfully reduced the LLOC by about 68% in this experiment.

5.4 Experiment 4

To evaluate the reduction of maintenance cost and investigate RQ4, we changed the design of the web application in Experiment 3 to modify the DOM structure. POGen makes manual changes to the test code unnecessary because the accessor methods depend on the names of template variables and not on the structure of the HTML document.

List 1.8. HTML templates of the result page with ejs before and after changing the design of the web application

```
<!-- Before changing the HTML template of the result page -->
<div><%= result %></div><br />
actual = [<span><%= out %></span>]<br />
expect = [<span><%= ex %></span>]
<!-- After changing the HTML template of the result page-->
<p> Your answer is <b><%= result %></b>!  </p>
<p> Your program's ouput is [<span><%= out %></span>].  </p>
<p> Then, our expected ouput is [<span><%= out %></span>].  </p>
```

List 1.8 shows the HTML template of the result page with ejs before (top) and after (bottom) changing the design of the web application. The generated skeleton test code provided the accessor methods with the same signatures. For example, the `getTextOfResult` method returned the string representation of the template variable result both before and after the change. If the names of the template variables are changed, then the test code must be changed manually due to changes in the names of the generated accessor methods. As we investigated web applications in a company, template variable name changes, however, occurred much less than structural changes to HTML documents. Roest et al. also [6] claim XPath for selecting DOM elements causes fragile tests. Therefore, POGen reduce maintainance costs by improving maintainability of test code.

6 Limitations

Dependency on template engines: Our technique cannot elucidate the dynamic components which are created using web frameworks or DOM API without template engines. However, many web frameworks such as Struts and Ruby on Rails have template engines, and developers typically use template engines.

Regeneration of accessor methods for template variables: When a template variable occurs more than once in an HTML template, POGen names the corresponding accessor methods differently with sequential numbers. Therefore, the names of accessor methods are changed when the same template variables are added into or removed from the HTML template, and the test code must be changed manually.

Assessing input values for testing: Template variable coverage cannot assess input values themselves. For example, template variable coverage does not change when the value "2" is used instead of "two" for {`$right`}. However, this limitation are common in structural test coverage criteria.

7 Related works

Staats et al. [7] claim that one should not only refer to the test coverage but also to the test oracles when discussing test quality. Schuler et al. [8] propose a new coverage criterion called checked statement coverage, which enhances existing statement coverage for white-box testing in terms of test oracle quality. Template variable coverage also considers both test coverage and test oracles, and therefore, our technique can enhance test quality for web applications.

Kodaka et al. [9] provide a tool for determining if the dynamic text generated by JSP is equal to the text expected by users. In contrast, POGen generates accessor methods for both HTML elements and texts containing template variables. Thus, our technique can test web applications with greater flexibly.

Mesbah et al. [10] propose a new technique for testing web applications with invariants and their crawler which supports AJAX user interfaces. Roest et al. also [6] propose a new technique for extracting patterns for invariants from dynamic contents such as tables and lists. Whereas their approaches generates invariants, which are independent on test scenarios, our approach generates accessors for template variables, which are dependent on test scenarios.

There are many researches of model based testing for web applications and GUI applications such as [11], [12] and [13]. Their approach only generate test scenarios without test oracles and cannot treat the web pages which do not appear in models. POGen, on the other hand, helps testers to write test code including test oracles. Thus, our technique can work well with conventional methods and conduct test web applications flexibly and reasonably.

8 Summary and future works

In this paper, we elucidated the problems in existing testing methods through motivating examples. We proposed a novel coverage criterion called template

variable coverage, as well as a novel technique to improve the template variable coverage with a tool called POGen. POGen generates skeleton test code, which includes accessor methods for the dynamic components of web applications, by analyzing HTML templates. Moreover, we evaluated the effectiveness of the template variable coverage and POGen in empirical experiments.

In the future, we will evaluate our approach and template variable coverage for real-world web applications with the mutation testing tools specific to web applications. We will also propose a new set of coverage criteria based on existing coverage criteria, which will target branches in HTML templates and production code on the server side to evaluate test quality from various viewpoints.

References

1. Choudhary, Shauvik Roy, Prasad, Mukul R. and Orso, Alessandro: CrossCheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications. *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST 2012)*, 171–180 (2012).
2. Jupiter Consulting: EJS - JavaScript Templates, `http://embeddedjs.com/`.
3. Oracle: JavaServer Faces, `http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html`.
4. Google: Closure Tools – Google Developers, `https://developers.google.com/closure/templates/`.
5. Simon Stewart : PageObjects, `http://code.google.com/p/selenium/wiki/PageObjects`.
6. Roest, Danny and Mesbah, Ali and Deursen, Arie van: Regression Testing Ajax Applications: Coping with Dynamism, *Proceedings of the Third International Conference on Software Testing, Verification and Validation*, pp. 127–136 (2010).
7. Staats, Matt and Whalen, Michael W. and Heimdahl, Mats P.E.: Programs, tests, and oracles: the foundations of testing revisited, *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pp. 391–400 (2011).
8. Schuler, David and Zeller, Andreas: Assessing Oracle Quality with Checked Coverage, *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST 2011)*, pp. 90–99 (2011).
9. Kodaka, Toshihiro, Uehara, Tadahiro, Katayama, Asako and Yamamoto, Rieko: Automated testing for Web applications using Aspect-oriented technology, *IPSJ SIG Notes 2007(33)*, pp. 97–104 (2007).
10. Mesbah, Ali and van Deursen, Arie: Invariant-based automatic testing of AJAX user interfaces, *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pp. 210–220 (2009).
11. Andrews, Anneliese A., Offutt, Jeff and Alexander, Roger T.: Testing Web applications by modeling with FSMs, *Software and System Modeling*, pp. 326–345 (2005).
12. Sprenkle, Sara and Cobb, Camille and Pollock, Lori: Leveraging User-Privilege Classification to Customize Usage-based Statistical Models of Web Applications, *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST 2012)*, pp. 161–170 (2012).
13. Mariani, Leonardo and Pezze, Mauro and Riganelli, Oliviero and Santoro, Mauro: AutoBlackTest: Automatic Black-Box Testing of Interactive Applications, *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST 2012)*, pp. 81–90 (2012).