WSEAS TRANSACTIONS on
INFORMATION SCIENCE & APPLICATIONS
Manuscript received Aug. 20, 2007; revised Nov. 20, 2007

Customizable Pattern-Oriented Verifiers for Web Applications

Nhor Sok Lang
University of Tokushima
Course of Info Sci. & Intel. Syst.
Tokushima
JAPAN
soklang@is.tokushima-u.ac.jp

Takao Shimomura
University of Tokushima
Dept. of Info Sci. & Intel. Syst.
Tokushima
JAPAN
simomura@is.tokushima-u.ac.jp

Quan Liang Chen
University of Tokushima
Course of Info Sci. & Intel. Syst.
Tokushima
JAPAN
quan@is.tokushima-u.ac.jp

Kenji Ikeda
University of Tokushima
Dept. of Info Sci. & Intel. Syst.
Tokushima
JAPAN
ikeda@is.tokushima-u.ac.jp

Abstract: This paper presents a method that enables programmers to easily define and customize verification items for Web applications. In this method, we first specify verification conditions using regular expressions to indicate a part of code of interest, and if it is against a verification item, this method provides a mechanism to automatically correct that code using another expression. In addition, to make it possible to easily define complicated verification items that check a sequence of the pieces of program code that appear at multiple locations of the programs, the method provides several program interfaces with which we can check the part of the program code of interest in more detail.

Key–Words: Customizable verifier, Regular expression, Verification, Web applications

1 Introduction

In order to develop Web applications of high quality, it is important to apply efficient frameworks to standardize the process of development in projects [1], [2], [3], [4], or apply useful design patterns to produce the program code that can easily be enhanced [5]. However, it is not enough. In addition to these efforts, we have to check the programs to see whether they keep various kinds of rules such as verification items for security which are common for all kinds of Web applications [6], [7], and verification items for coding styles or code conventions which are pertain to each project [8].

Tools that assist these kinds of verification are often incorporated into a part of integrated development environments [9]. However, in most of the existing tools [10], [11], [12], [13] , [14] , [15] , [16] , their verification items are determined in advance, or even if verification items can be added, they only permit addition of the conditions that match a part of program code to be verified. They do not have any mechanisms that help programmers correct the part of program code if it turns out to be against the rules. Moreover, it is difficult to add verification items that check a sequence of the pieces of program code that appear at multiple locations of the programs.

This paper presents a method that enables programmers to easily define and customize verification items. In this method, we first specify verification conditions using regular expressions to indicate a part of code of interest, and if it is against a verification item, this method provides a mechanism to automatically correct that code using another expression. In addition, to make it possible to easily define complicated verification items that check a sequence of the pieces of program code that appear at multiple locations of the programs, the method provides several program interfaces with which we can check the part of the program code of interest in more detail [17].

2 Requirements for Verifiers

2.1 Verification items

For security, consistency, or completeness of Web applications, we need to check various kinds of items as follows:

- Continuous sessions

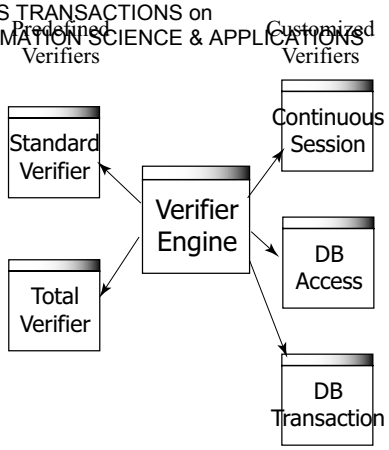

Figure 1: Window configuration of Verifier engine

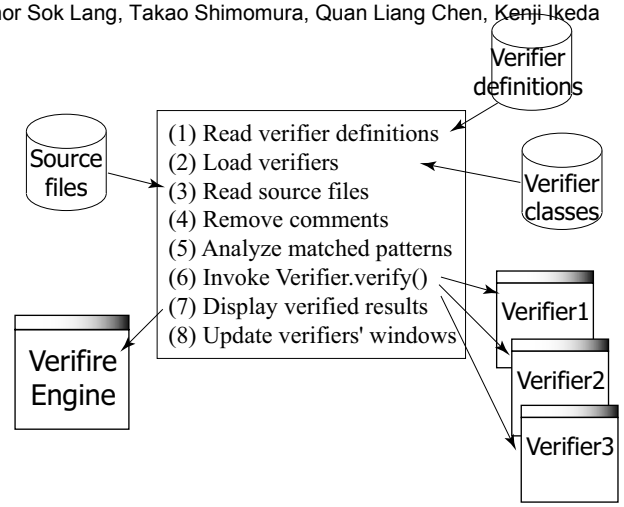

Figure 2: Verification Processes

- Connection and transaction processes for database
- Mutual exclusion of Servlets
- Secrecy of passwords
- SQL injections
- Cross-site scripting
- Authentication of users
- URL encoding of HTTP requests
- Consistency of parameters of form data to be sent and received
- Internationalization of messages and images

For example, as for the verification item of continuous sessions, the URL of a Web page control transfers to should be written as follows to continue a session:

response.encodeURL(”example.jsp”)

When we simply write the URL of the next Web page as ”example.jsp”, if a Web browser is set not to accept cookies, which is sent from a Web server, a user’s session will not be properly continued. In the program code of Web applications, these URL appear in such locations as:

<a href=”example.jsp” ...>
<form action=”example.jsp” ...>
window.open(”example.jsp”, ...)

The first <a> tag appears at a link of a Web page. When we click on the link, the next Web page will be shown. The second <form> tag is used when we enter some data and submit them. The third window.open function appears in a part of JavaScript code. For example, when we click on a button, its onclick event handler will be invoked. This handler will invoke the window.open function to open a new Web browser window or update an existing Web browser window. If the URL of a Web page control transfers to is not written in such a way that the session will continue, we would like to detect such code, and automatically replace the URL in the code with the correct code wrapped by response.encodeURL() method.

2.2 Requirements

To make it possible to flexibly and easily verify Web applications, we take into account the following requirements:

Requirement 1 We can easily add, modify and delete verification items.

Requirement 2 If a part of program code is against a verification item, the part of program code can be automatically corrected.

Requirement 3 In addition to verifying a continuous part of program code, the relationship between several parts of program code that appear at distinct locations of the program can also be verified.

Methods	Functions
int patternCount()	Number of pieces of matched text
String getText(int patternIndex)	patternIndex-th matched text
int getLineNo(int patternIndex)	Line no. of patternIndex-th matched text
int getLineOffset(int patternIndex)	Column no. of patternIndex-th matched text
int groupCount(int patternIndex)	Number of groups of patternIndex-th matched text
String getGroupText(int patternIndex, int groupIndex)	Text of groupIndex-th group of patternIndex-th matched text
void addProblem(int patternIndex)	Mark patternIndex-th matched text
void addMessage(message)	Display message in Verifier engine

Table 1: Verification Methods for Customization

3 Verifier Engine and Verifiers

3.1 Window configuration of verifier engine and verifiers

As shown in Fig. 1, the system Verifier engine opens a verifier window for each of verification items. The window displays the verification result of the verification item and assists automatic correction of a part of program code that is against the verification item. Immediately a part of the program code is automatically corrected in one verifier window, the contents of all the other verifier windows will be updated to reflect this correction. This system provides Standard Verifier and Total Verifier as predefined verifiers. In Standard Verifier, programmers can enter a verification condition and see the verification results right away. A verification condition is specified as a regular expression, which could be complicated. Standard verifier helps understand exactly which pattern will match a verification condition, which is entered as a regular expression. When we construct a customized verifier class, we can also use this Standard Verifier to see whether its verification conditions are valid or not in advance. Total Verifier shows the verification results of all the verifiers that are registered in the Verifier engine at a time. The displayed results may be complicated because the verification results of multiple verification items are displayed at a time. However, it is convenient when we check whether the verification of all verification items is completed or not after we finish verifying a series of verification items.

3.2 Verification processes

Figure 2 shows the verification processes of Verifier engine. The system first (1) reads a verifier definition file, which defines a verification condition and a verifier class'es name, and then (2) dynamically loads the verifier classes defined in the file. Next, according to a user's operation, (3) it reads a source program file to be verified. (4) It then removes comments from the source code to obtain the text to be verified, and concatenates all source lines so that it can verify a part of program code that exists in multiple lines. When a verification item is selected, the system creates an instance of the verifier class that corresponds to the selected verification item. (5) It analyzes parts of program code (the code of interest) each of which matches the verification condition specified in the verification item. (6) The system invokes verify() method of the instance of the verifier class. This verify() method checks whether the matched parts of program code are correct or wrong. (7) The system displays the summary of the verification results returned by the verifier in the Verifier engine window. (8) The verifier updates the display of the verifier window based on the verification results.

4 Implementation

4.1 Verification methods for customization

For each verification item, Verifier engine creates a sequence of the pieces of the code of interest, each of which matches the verification condition. The verifier class of each verification item performs its own verification using this sequence of code. To make it easier to verify the code, Verifier engine provides public methods shown in Table 1 for each verifier class. For example, to know the number of pieces of text that match a verification condition, patternCount() method is invoked. Using the value "numOfPatterns" returned by this method, each matched text can be obtained by invoking getText(index) method, where index ranges from 0 to "numOfPatterns" - 1. The verifier class uses these public methods to check whether the sequence of the code of interest is correct or wrong, and informs the Verifier engine of the result using addProblem() method. It can also send the summary of the verification results to the Verifier engine using addMessage() method.

4.2 Definition of verification items

To make it possible to easily add verification items, we prepare a file (verifier definition file) to define verification items (Requirement 1). Using the verifier definition file, we define the name of a verification item, the name of a verifier class, a verification condition, and a method of correction as shown in Fig. 3 (a). We use a regular expression to define the verification condition. For example, in the verification of continuous sessions described in Section 2.1, we define the code, in which the URL of a Web page control transfers to appears, using a regular expression "(window.open\(|action=|href=)"(.+?)"" as shown in Fig. 3(a).

To make it possible to automatically correct code when it is against a verification item, we provide a method to specify how to correct the code of interest (Requirement 2). We first divide the part of the code that matches the verification condition, which is written using a regular expression, into several groups. Using symbols $1, $2, $3, ... , each of which indicates the group number of the group, we specify the correct code which will be replaced with the code of interest. For example, in the verification of continuous sessions described in Section 2.1, we specify the correct code as "$1"<%= response.encodeURL("$2") %>"" as shown in Fig. 3(a). For example, suppose that a part of source text "window.open("/sp/request1.jsp"" matches the verification condition. This source text is divided into two groups, "window.open(" and ""/sp/request1.jsp"". The first group is referred to as $1, and the second group is referred to as $2. Therefore, the correction of code will be "window.open("<%= response.encodeURL("/sp/request1.jsp") %>"".

We define a verifier class so that we can also verify the relationship between several parts of program code that appear at distinct locations of the program in detail (Requirement 3). We define this verifier class to be a class that extends abstract class Verifier. The verifier class can perform detailed verification using verification methods for customization shown in Table 1. For example, in the verification of continuous sessions described in Section 2.1, when the code of interest does not include "response.encodeURL", this verifier class judges that the code is wrong, and informs Verifier engine using addProblem() method that the code of interest has a problem. It also informs Verifier engine of the number of the pieces of wrong code using addMessage() method. We apply the template method [5] to verifier classes because most of the processes are common among them. Such common processes are implemented in abstract class Verifier. By extending abstract class Verifier, each verifier class has only to define its own particular verification using verify() method as shown in Fig. 3 (b).

(a) Verifier Definition

```
Continuous Session
ContinuousSession
(window.open\(|action=|href=)"(.+?)"
$1"<%= response.encodeURL("$2") %>"
```

(b) Verifier Class

```
public class ContinuousSession extends Verifier {
  public void verify() throws Exception {
    int numOfPatterns = patternCount();
    for (int i = 0; i < numOfPatterns; i++) {
      String pattern = getText(i);
      Pattern p = Pattern.compile(".*response.encodeURL.*");
      Matcher m = p.matcher(pattern);
      boolean isNoProblem = m.matches();
      if (!isNoProblem) addProblem(i);
      .....
      addMessage("Number of problems = " + numOfProblems);
      .....
}
```

Figure 3: Definition of verified items and their verification classes

4.3 System configuration

Figure 4 shows the relationship between a verifier class for each verification item and the abstract class Verifier that each verifier class extends. (1) When a verification item is selected from a menu of Verifier engine window, the system loads the corresponding verifier class verCls, and creates an instance ver of this verifier class. Then, the display() method of the instance ver is invoked, and this actually executes the display() method of the abstract class Verifier. This method detects the pieces of code that matches the specified verification condition of the verification item, and then, invokes the verify() method of instance ver to let it verify the program code according to its verification item. The method finally displays the verification results in the verifier window. In a verifier window, (2-1) as soon as a part of code is corrected, the document in instance ver (actually, the document defined in the abstract class Verifier) will detect this modification and update the source code stored in the verifiedText object. Then, the verifiedText object increments the value of variable textVersionNo by one, which represents the version number of the source code, and (3) informs all verifier classes of this modification to let them update the contents of their verifier windows. When a new source file is read, (2-2) it also informs all verifier classes of this change of source code to let them update the contents

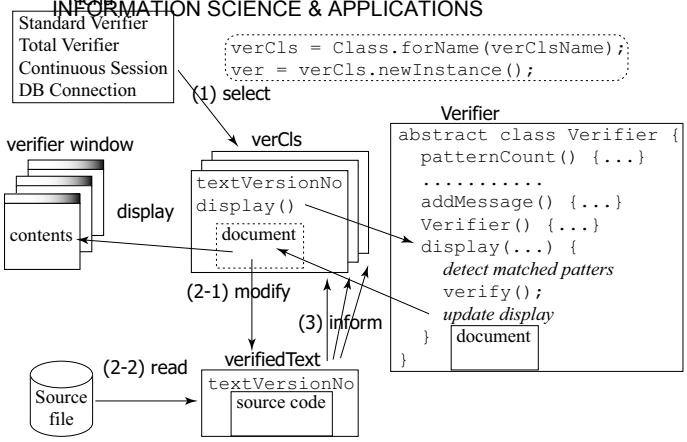

Figure 4: Relationship between each verifier and its abstract class

of their verifier windows. An instance of each verification class also has variable textVersionNo, and only if this value is consistent with the value of textVersionNo in the verifiedText object, the verifier instance updates the source code stored in the verifiedText object to update the contents of its verifier window. Otherwise, this window has already been updated when the source code is modified in one of the other verifier windows.

4.4 Correspondence between source and verified text

The source text is read from a source file, and it may include some comments, which is not the target of verification. The system removes these comments from the source text to obtain the text to be verified. It has to manage the correspondence of positions between them, because it will display the verification result on the source text. Fig. 5 shows how to manage the correspondence between source text and verified text. Suppose that the source text contains some lines, whose lengths are l1, l2, l3, and so on, and includes two parts of comments, the lengths of which are c1 and c2, respectively. After comments are removed from the source text, the system obtains the verified text, whose line lengths are l1, l2, s1, l4, and l5 - c2, and so on. A two-dimensional array vs[][] records the cumulative length of each comment's position in both of the verified and source text. For example, the first comment starts at a position of l1 + l2 + s1 in the verified text, while it ends at a position of l1 + l2 + l3 in the source text. s(v) function returns a position in the source text that corresponds to a position v in the verified text. isV(s) function returns whether a position s in the source text is included in the verified

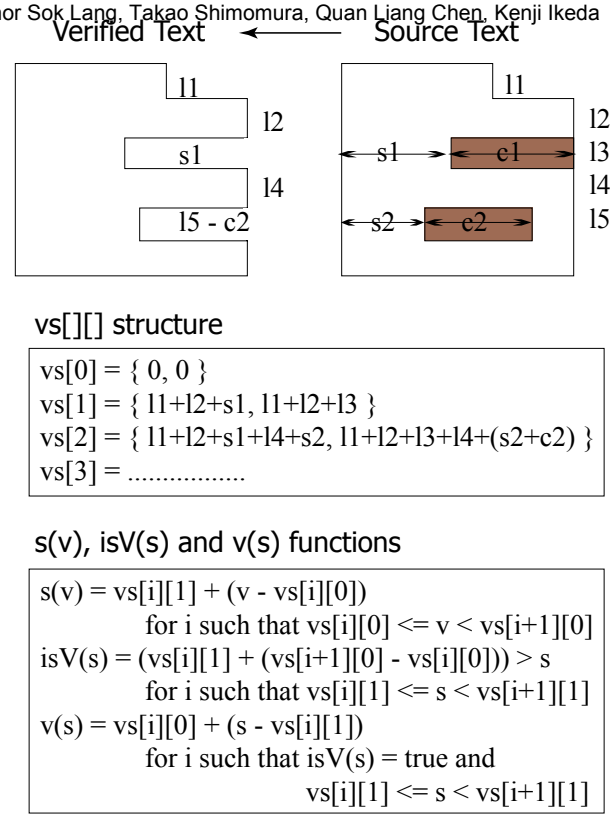

Figure 5: Correspondence between source and verified text

text. If isV(s) is true, v(s) function returns a position in the verified text that corresponds to a position s in the source text. For example, let v be a position of l1 + l2 + s1 + 1 in the verified text. Because vs[1][0] <= v < vs[2][0], s(v) will be vs[1][1] + (v - vs[1][0]) = l1 + l2 + l3 + 1. Reversely, let s be a position of l1 + l2 + l3 + 1 in the source text. Because vs[i][1] <= s < vs[2][1], isV(s) will be (vs[1][1] + (vs[2][0] - vs[1][0])) > s, which is l1 + l2 + l3 + l4 + s2 > l1 + l2 + l3 + 1. It becomes true. Then, v(s) will be vs[1][0] + (s - vs[1][1]) = l1 + l2 + s1 + 1.

4.5 Example of continuous sessions

If we register verifier class Continuous Session shown in Fig. 3, which performs a verification of continuous sessions, in Verifier engine, we can perform verification as shown in Fig. 6. We here use a simple source program of JSP page "verifier.jsp" as an example [18]. In the verifier window of Continuous Session, five pieces of code are detected as the URL of a Web page control transfers to. The pieces of detected code are shown in yellow. Among them, the pieces of wrong code are shown in blue. The wrong code is the code that Continuous Session verifier judged to be

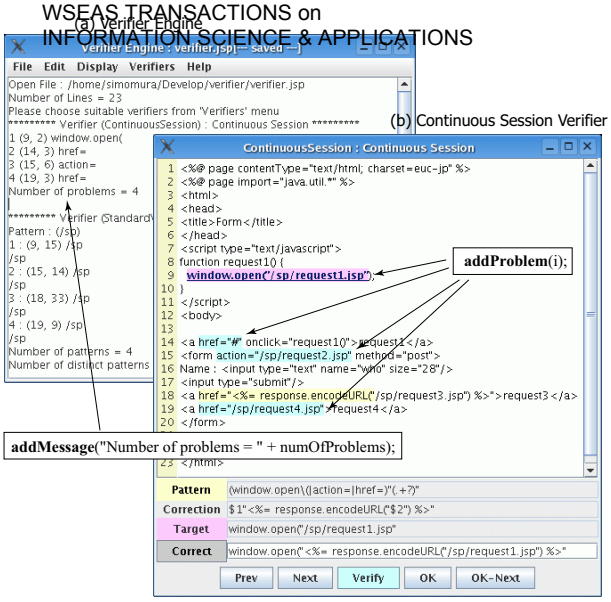

Figure 6: Verification of continuous sessions

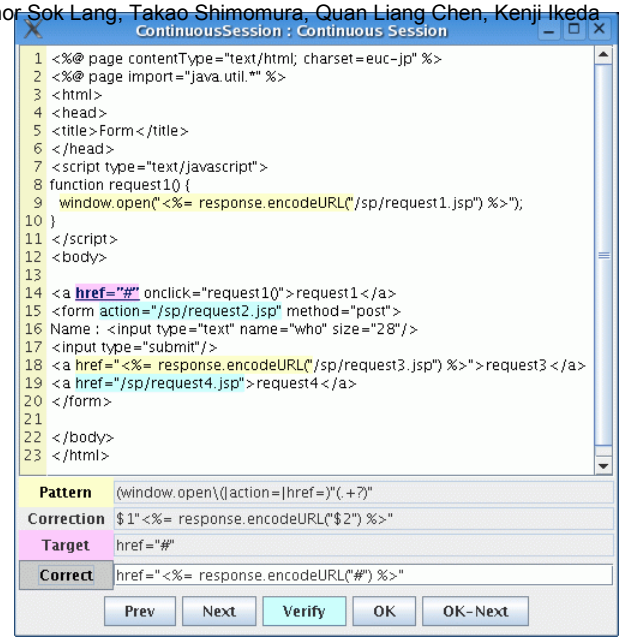

Figure 7: Correction of continuous sessions

wrong and informed Verifier engine of using addProblem() method. By clicking the "Next" button, we can traverse the pieces of wrong code in turn. Among them, which are shown in blue, the current wrong code is shown in red with an underline. In Fig. 6 (b), the part of code "window.open("/sp/request1.jsp"" indicates the current wrong code. If we click the "Correct" button, this part of code will be automatically corrected to "window.open("<%= response.encodeURL("/sp/request1.jsp") %>"" (See Fig. 8 (a)). If we click the "Verify" button, Verifier engine window will show the summary of the verification results that Continuous Session verifier informed Verifier engine of using addMessage() method (See Fig. 6 (a)).

4.6 Standard verifier

Figure 7 shows the result after we verify the target program "verifier.jsp" using Continuous Session verifier and then perform automatic correction on the piece of wrong code in line 9. Figure 8 shows the result after we verify the same program using Standard Verifier. In its verifier window, we check the parts of code that matches "/sp". Although the verification results are different in these two verifier windows, both windows always show the same contents of the program even if it is modified.

In the Standard Verifier, when we specify a verification condition, we can immediately see the pieces of code the Standard Verifier detects. When we construct a customized continuous session verifier class, we can know what parts of code will match the regular expression "(window.open\(|action=|href=)"(.+?)"" in advance. Moreover, we can also know which part of code will match a more complicated regular expression "(<input [^>] +?type="password" [^>] +?value=") ([^"] *?)(")". If we need to modify the program code a little bit, we can do it using Standard Verifier without constructing a verifier class. When we click the "Verify" button, Standard Verifier shows the total number of the pieces of code that match the verification condition, and the number of the pieces of distinct code among them (See Fig. 6 (a)).

5 Observation

This system enables us to easily add a new verification item. We do not need to write a lot of code to verify the item. Table 2 illustrates verification conditions, correction code, and the lines of code which is required to define a verifier class for some verification items. For example, in the verification of continuous sessions described in Section 2.1, we were able to define the verification class in 16 lines of code. In addition, this system made it possible to easily verify a sequence of the pieces of program code that appear at multiple locations of the programs such as connection and transaction processes for database. For the database connection, it verified the correct sequence of code for open and close, such that "con = DriverManager.getConnection()" —> "stm = con.createStatement()" —> "stm.close()"

Table 2: Definitions for Customizable Verifiers

Verification Item	**Condition**	**Correction**	**Verifier Class** (lines of code)
Continuous session	(window.open\(|action=|href=)”(.+?)”		16
	$1”<%= response.encodeURL(”$2”) %>”		
DB connection	getConnection|createStatement|stm.close|con.close	$0	25
DB transaction	setAutoCommit|stm.exec|commit|rollback	$0	30
URL encoding	\?[a-zA-Z0-9]+?=.+?[”;\n]	$0	16
Password secrecy	(<input [^>] +?type=”password” [^>] +?value=”) ([^"] *?)(”)	$1$3	14

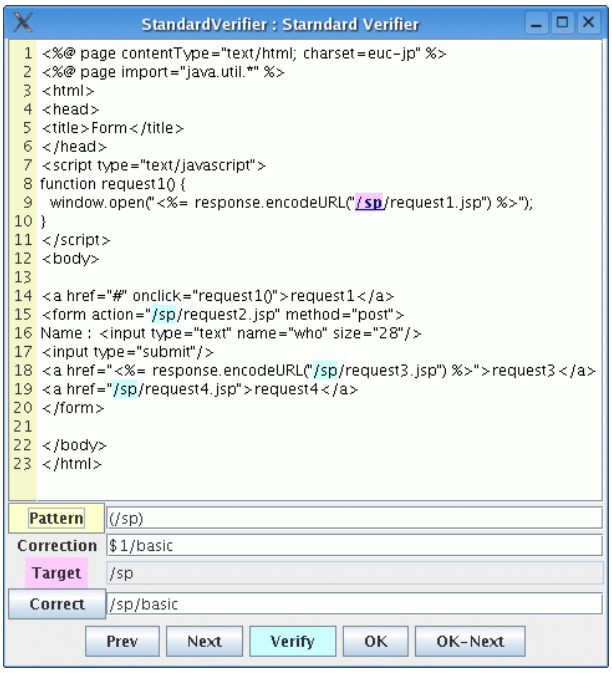

Figure 8: Standard verifier

—> “con.close()”. For the database transaction, it verified the correct sequence of code for commit and rollback, such that “con.setAutoCommit()” —> “stm.executeUpdate()” or “stm.execute()” —> “con.commit()” —> “con.rollback()”. Moreover, it verified whether the program locks the database tables in total order to prevent dead lock.

6 Conclusion

This paper has presented a method that makes it possible to verify the program code that implements Web applications from various points of view using various kinds of verification items. Customized verification items can be accumulated as not only the know-how for Web application development but also a part of executable verification mechanism. In future, we are going to introduce a meta-language for regular expressions to enable programmers to easily define their verification items and verifier classes.

References:

[1] *The Apache Software Foundation : Struts*. http://jakarta.apache.org/struts/, 2004.

[2] A. S. Christensen, A. Moller, and M. I. Schwartzbach. Extending java for high-level web service construction. *ACM TOPLAS*, Vol. 25, No. 6, pp. 814–875, 2003.

[3] Takao Shimomura. Visual design and programming for web applications. *Journal of Visual Languages and Computing*, Vol. 16, No. 3, pp. 213–230, 6 2005.

[4] A. Leff and J.T. Rayfield. Web-application development using the model/view/controller design pattern. pp. 118–127, 9 2001.

[5] Steven John Metsker and William C. Wake. *Design Patterns in Java*. Addison-Wesley, 4 2006.

[6] Takao Shimomura, Kenji Ikeda, Quan Liang Chen, Nhor Sok Lang, and Muneo Takahashi. Apty: Easy enjoyable effective e-learning. In *Proc. of the 7th WSEAS International Conference on Applied Informatics and Communications*, pp. 211–216, 8 2007.

[7] Takao Shimomura, Quan Liang Chen, Nhor Sok Lang, and Kenji Ikeda. Integrated laboratory network management system. In *Proc. of the 7th*

WSEAS International Conference on Applied Informatics and Communications, pp. 188–193, 8 2007.

[8] *Sun Microsystems, Inc. : Code Conventions*. http://java.sun.com/docs/codeconv/, 2006.

[9] *The Eclipse Foundation : Eclipse*. http://www.eclipse.org/, 2006.

[10] D. Castelluccia, M. Mongiello, M. Ruta, and R. Totaro. Waver: A model checking-based tool to verify web application design. *Electronic Notes in Theoretical Computer Science*, Vol. 157, No. 1, pp. 61–76, 5.

[11] E. Di Sciascio, F.M. Donini, M. Mongiello, and G. Piscitelli. Anweb: a system for automatic support to web application verification. pp. 609–616, 7.

[12] Marco Pistore, Marco Roveri, and Paolo Busetta. Requirements-driven verification of web services. *Electronic Notes in Theoretical Computer Science*, Vol. 105, No. 10, pp. 95–108, 12.

[13] Francesco Maria Donini, Marina Mongiello, Michele Ruta, and Rodolfo Totaro. A model checking-based method for verifying web application design. *Electronic Notes in Theoretical Computer Science*, Vol. 151, No. 2, pp. 19–38, 5.

[14] L. de Alfaro. Model checking the world wide web. pp. 77–85, 2001.

[15] Shriram Krishnamurthi. Web verification: Perspective and challenges. *Electronic Notes in Theoretical Computer Science*, Vol. 157, No. 2, pp. 41–46, 5.

[16] D.R. Licata and S. Krishnamurthi. Verifying interactive web programs. pp. 164–173, 2004.

[17] Takao Shimomura, Kenji Ikeda, Chen Liang Quan, Lang Sok Nhor, and Takahashi Muneo. Customizable verifiers for web applications and their implementation. In *Proc. of WSEAS International Conference on Computer Engineering and Applications*, pp. 396–401, 1 2007.

[18] *Sun Microsystems, Inc. : JavaServer Pages Technology*. http://java.sun.com/products/jsp, 2006.