Lowering the Barriers to Website Testing with CoTester

Jalal Mahmud and Tessa Lau
IBM Almaden Research Center
650 Harry Rd,
San Jose, CA 95120, USA
{jumahmud, tessalau}@us.ibm.com

ABSTRACT
In this paper, we present CoTester, a system designed to decrease the difficulty of testing web applications. CoTester allows testers to create test scripts that are represented in an easy-to-understand scripting language rather than a complex programming language, which allows tests to be created rapidly and by non-developers. CoTester improves the management of test scripts by grouping sequences of low-level actions into subroutines, such as "log in" or "check out shopping cart", which help testers visualize test structure and make bulk modifications. A key innovation in CoTester is its ability to automatically identify these subroutines using a machine learning algorithm. Our algorithm is able to achieve 91% accuracy at recognizing a set of 7 representative subroutines commonly found in test scripts.

ACM Classification Keywords
H.3.3 Information Systems: Search and Retrieval; H.5.2 Information Interfaces and Presentation: User Interfaces

General Terms
Algorithms, Design, Human Factors, Experimentation

Author Keywords
Website Testing, Test Script, Instruction, Subroutine

INTRODUCTION

The web has become an indispensible part of our daily activities. As more and more applications move to the web, there is a growing need for tools to assist with web application development and testing. However, today's web testing tools, such as Rational's Functional Tester [3] and HP's Mercury [2], present several barriers to use. First, they require programming ability: tests are recorded in programming languages such as Java or Visual Basic. Second, maintaining a corpus of tests can be time-consuming, particularly during iterative development as applications change and tests need to be kept in sync. As a result of these barriers, testing tools are not as widely used as they could be.

IUI 2010, February 7 - 10, 2010, Hong Kong, China.

CoTester, which is built on the CoScripter [17, 16] platform, provides testers with features specifically targeted at functional web testing, including test management and support for assertions. By applying CoScripter's easy-to-understand scripting language (ClearScript) to the domain of web testing, CoTester enables testers with a wide variety of skill levels to create and maintain test scripts. CoTester extends the ClearScript language with support for assertions [18], which are an important aspect of functional testing.

Once a set of tests has been created, they may need to be updated frequently as the application under test (*e.g.*, a website) changes. Updating tests may require much manual effort by testers, who have to manually inspect each test and make the required changes to update it for the new application. Global search and update is typically not sufficient in these cases. For example, a web application may change its checkout process such that a new checkbox "notify by email" is added to the checkout page. If the checkout process could be automatically identified, a tester would be able to easily add an assertion to test the presence of this checkbox across all instances of this process. As another example, a tester may want to add an assertion after the login process to check that a user has successfully logged in to the website. This can not be done automatically unless the login process is identified from the test scripts.

Subroutines group together a sequence of low-level actions within a test (*e.g.*, entering a username, entering a password, and clicking on a "Sign in" button) into a conceptual unit representing a higher-level action, such as "Log in to the website". They enable better test management and help testers visualize test structure. For example, Figure 2 shows the subroutines identified from the scripts in Figure 1. Subroutines have the promise to make test maintenance easier by enabling testers to automatically apply a similar change to all instances of a subroutine across all test scripts. For example, the instruction *check the "notify by email" checkbox* in a checkout subroutine could be added automatically across all the test scripts which have a checkout process. In addition, assertions [18] could be automatically added to the start/end of each subroutine to ensure that certain conditions hold before/after every instance of the subroutine (*e.g.*, asserting that the user's name is displayed on the page after a login has been completed, asserting that a radiobutton "delivery method" is present before the checkout).

- go to "http://www.southwest.com"
- click the "MySouthwest Login" link
- enter "678899532" into the "Account Number" textbox
- enter your password into the second textbox
- turn on the "Remember my account number for future login." checkbox
- click the "Login" button
- assert there is an element containing "Account"
- click the "Online Checkin" link

(a)

- go to "http://www.southwest.com"
- click the "Book Travel" link
- assert there is an element containing "My Southwest Login"
- enter "456712345" into the "Account Number" textbox
- assert there is a "Password" textbox
- enter your password into the "Password" textbox
- click the "Login" button
- assert there is an element containing "Account Snapshot"

(b)

Figure 1. Example test scripts for two tasks on southwest.com

To help testers maintain large corpora of test scripts, we have designed and implemented a machine learning algorithm to *automatically identify subroutines* in test scripts. Using apriori labelled samples of subroutines collected from a number of scripts, we learn models of subroutines. Such models are used to automatically recognize subroutines within a test script.

In this paper, we make the following contributions:

- An implemented system, CoTester, which builds on CoScripter [17, 16] to provide a lightweight, easy-to-use system for web test automation.
- Extensions to the ClearScript language for representing assertions, which are fundamental for functional testing.
- A machine learning algorithm for automatic subroutine identification from test scripts.
- An empirical evaluation of our algorithm, showing that it is capable of recognizing a set of 7 subroutines with 91% accuracy when the algorithm is trained and tested on scripts recorded from the same website.

RELATED WORK

Here we present prior work related to our contributions.

Website Testing Tools

There are a number of commercial and open source tools available which assist in the automation of web testing [3, 21, 1]. Most of the commercial testing tools (*e.g.*, SilkTest [1], Rational Functional Tester (RFT) [3]) are made for website developers and testers with programming knowledge. For example, SilkTest [1], developed by Borland Software, uses the proprietary 4Test language for automation scripting. Rational Functional Tester (RFT) [3] records test scripts in the Java language. HP's QuickTest [2] records test scripts

* go to "http://www.southwest.com"

Login

  * click the "MySouthwest Login" link
  * enter "678899532" into the "Account Number" textbox
  * enter your password into the second textbox
  * "turn on the "Remember my account number for future login." checkbox
  * click the "Login" button
  * assert there is an element containing "Account"

* click the "Online Checkin" link

(a)

* go to "http://www.southwest.com"
* click the "Book Travel" link

Login

  * assert there is an element containing "My Southwest Login"
  * enter "456712345" into the "Account Number" textbox
  * assert there is a "Password" textbox
  * enter your password into the "Password" textbox
  * click the "Login" button
  * assert there is an element containing "Account Snapshot"

(b)

Figure 2. Subroutine identified from scripts in Figure 1. Instructions grouped together as a subroutine are shown as right indented under the subroutine.

written in the Visual Basic language. Other functional testing tools include Selenium [21], Sahi [19], Concordian [7].

However, test scripts recorded by these tools require some amount of programming knowledge to be able to understand and edit them. In contrast, by leveraging CoScripter's [17, 16] easily understandable scripting language, CoTester enables testers without programming ability to create and edit test scripts. In addition, CoTester can improve test script maintanence by identifying the subroutines from test scripts. Most often developers do not use functional testing until the application is mostly complete, since the application changes too often during the development phase. Subroutine detection could improve test script maintanence during such a phase and thus enable testing during the iterative development of an application.

End User Programming and Task Learning

Subroutine identification from test scripts is related to research on programming by demonstration [8, 15], and task learning [4, 5, 6, 13, 10, 22, 12].

Programming by demonstration [8, 15] allows users to construct a program by simply performing actions in the user interface, with which they are already familiar. For example, Eager is a PBD system that observes a user executing a task one or more times and then infers a general procedure to do the task [8]. However, it assumes a fixed structure of the task (fixed number of steps, e.g., steps to fill out a form) and can generalize only if the user repeats the steps (e.g., a user may demonstrate the same sequence of steps multiple times and the system infers a general procedure). PBD trace generalization using a machine learning technique is described by Lau and Weld [15]. However, such generalization also learns a single pattern within a task and can not learn variability of the structure of the task, *e.g.*, different ways of doing a checkout in an e-commerce website, different ways of adding an item to a shopping cart. In contrast, subroutine learning learns conceptual units from already existing executable scripts. It does n't assume any fixed structure of the subroutine and can learn variance of the structure (i.e.

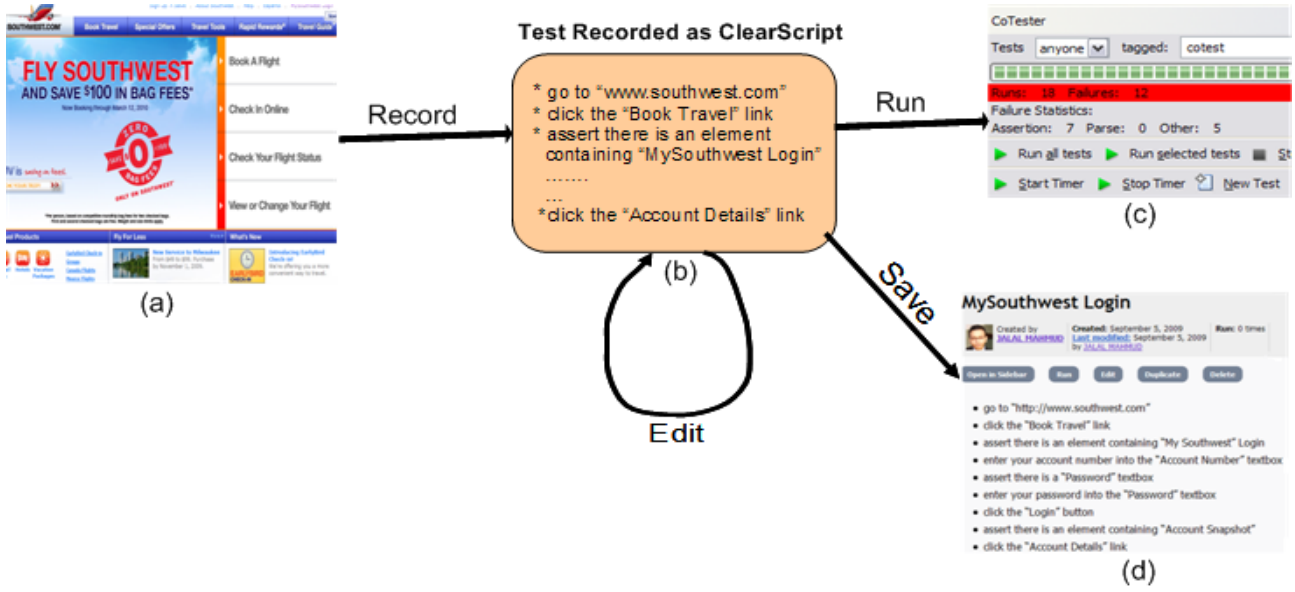

Figure 3. Testing using CoTester - (a) A webpage (b) Test script (c) CoTester sidebar (d) CoTester shared repository

sequence of instructions) of a subroutine from its instances collected from a number of test scripts. In addition, subroutine identification can identify one or more instances of subroutines from a test script, which is in contrast to PBD systems that can detect only one task at a time.

Task learning systems learn task models from users' examples. Huffman et al. illustrate how an intelligent agent can be taught to perform tasks [12]. Tailor [5] enables users to modify task information through instruction. A framework for learning hierarchical models of web service procedures is described in [6]. Sheepdog [13] learns procedures by demonstration by watching multiple experts performing the same procedure across different conditions. PLOW [4] is a collaborative task learning system that learns task models by demonstration, explanation, and dialog. Lapdog [10] learns procedures in emails from one or more examples. Spaulding et al. describe a task learning system that learns executable procedures from user demonstration and instruction [22].

Subroutine identification from test scripts is different from all of the above mentioned research in the following ways. First, task learning systems learn executable tasks from users actions. Their goal is to automate such tasks. Most of them use special semantics of a task (*e.g.*, precondition, postcondition) to learn the task models. In contrast, we learn conceptual units from already existing executable scripts to help testers visualize test structure and maintain test scripts. In addition, learning subroutines does not use the above mentioned special semantics. Second, most of the task learning systems require a lot of human interaction and language-rich demonstration from users as part of the learning process. In contrast, we try to minimize the amount of information user has to provide during the learning process.

However, the fundamental difference between our work and all of the above mentioned research is that subroutine identification from test scripts is focused on improving test script maintanence which is not the goal of the existing task learning systems. To the best of our knowledge, there is no prior research that automatically identifies such subroutines from scripts.

THE COTESTER SYSTEM

CoTester is built on top of CoScripter [17, 16] and allows testers to create test scripts that are represented in the ClearScript language, an easy-to-understand scripting language. To better support testing, we have extended the original ClearScript language used in CoScripter with assertions, which are used to test the presence or absence of elements on the page and are fundamental for functional testing. Table 1 shows examples of CoTester assertions.

Figure 3 illustrates a high level picture of the CoTester system. Figure 3 (a) shows the homepage of "www.southwest.com". The example test script of the picture is recorded from this website. The recording of such a script is done by doing actions on web pages, and generating an instruction for each of them (Figure 3 (b)). As users interact with the browser performing a process, CoTester records all the forms filled, links and buttons clicked, and generates instructions for each action. In addition, users can also insert assertions in a test script, and edit instructions. Assertions can be inserted in the following ways: i) manually editing the script, or ii) using our interface as illustrated in Figure 4. In Figure 4, a user first clicks the assert toolbar button of the sidebar. When the user is in assertion mode and moves the mouse pointer over a web page element, our system highlights that element by showing a red rectangle surrounding it. The user can click the highlighted element to insert an assertion. Therefore, highlighting and clicking "Manage Your Travel" inserts an assertion for this text in the script.

Type	Example
Presence of an Object	assert there is a link assert there is a button assert there is a checkbox
Absense of an Object	assert there is no link assert there is no radiobutton assert there is no listbox
Presence of an Object with a Caption	assert there is a “southwest” link assert there is a “go” button
Absense of an Object with a Caption	assert there is no “address” textbox assert there is no “Image Search” link
Presence of an Object with a Caption and a Value	assert there is a text “san jose” into the “city” textbox
Absense of an Object with a Caption and a Value	assert there is no “CA” into the “state” listbox
Presense of a Text	assert there is a td that contains “Please enter” assert there is a div that starts with “Almaden” assert there is an element that ends with “Click here”
Absense of a Text	assert there is no element that contains “Pay bill”

Table 1. CoTester Assertions

The CoTester user interface allows saving the recorded script in a centralized shared repository where a community of users can share, run and collaboratively develop test scripts (Figure 3 (d)). It also allows easy management of test scripts (Figure 3 (c)). Users can tag test scripts in the shared repository and select the scripts tagged by them, or anyone. They can run a single test script, or a batch of scripts. When a script is executed, each instruction (either assertion or regular instruction) is parsed and the Document Object Model [9] of the web page is analyzed to find the desired element. In case of a successful match of the element, the system highlights the matched element on the web page, and executes the instruction. However, in case of a non-match, the instruction is not executed and the test fails. If all such instructions are successful, we say that the result of the test is a “Success”. Otherwise, it is a “Failure”. When a test fails, the system outputs the reason for the failure (*e.g.*, could not find the element with label “Place Order”). Figure 5 shows the output of CoTester after a set of test scripts has been executed. After a batch of scripts have been run, users can save the test report as a spreadsheet (see Figure 6). In the next section, we will describe how subroutines from test scripts are identified.

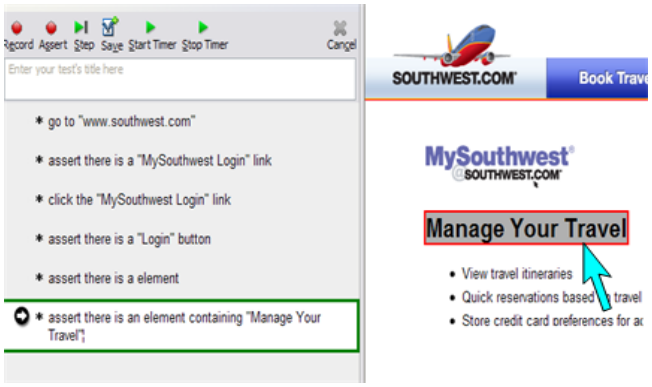

Figure 4. Inserting assertions to scripts using CoTester interface

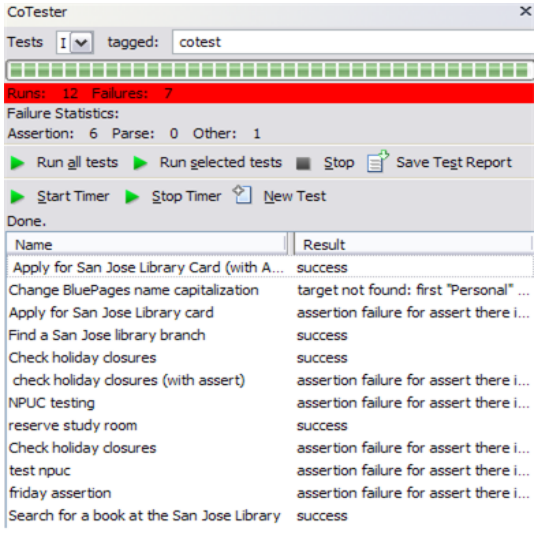

Figure 5. Test Management using CoTester

SUBROUTINE IDENTIFICATION FROM TEST SCRIPTS

Once a set of tests has been created, they may need to be updated frequently as the website changes. To reduce the manual effort to update test scripts, we have designed and implemented a machine learning algortihm to identify subroutines from test scripts. Our algorithm could make test maintenance easier by enabling testers to automatically apply a similar change to all instances of a subroutine across all test scripts. In this section, we describe our subroutine identification algorithm. First, we discuss a few preliminary concepts.

Technical Preliminaries

The *Vector Space Model* (VSM) is widely used in Information Retrieval systems for document retrieval [20]. In this model, a document is represented as a vector, where each dimension corresponds to a separate term. Typically terms are single words, bigrams, trigrams, or even longer text strings.

	A	B	C
1	total tests completed	4	
2	total tests succeeded	2	
3	total tests failed	2	
4	total tests failed for assertion	1	
5	total tests failed for parse	1	
6			
7	Individual Test Statistics		
8			
9	Test Title	TimeStamp (hh:mm:ss n	Test Result
10			
11	Apply for San Jose Library Card	4::07::8pm 7/ 23/ 2009	Assertion Failure
12			
13	Change BluePages name capitalization	4::07::9pm 7/ 23/ 2009	Success
14			
15	Apply for San Jose Library card	4::07::22pm 7/ 23/ 2009	Success
16			
17	Find a San Jose library branch	4::07::27pm 7/ 23/ 2009	Parse Failure

Figure 6. An example test report saved by CoTester

Each term appearing in the document is assigned a non-negative weight. One popular weighting scheme is TF*IDF [20]. It uses the following expression to assign weights:

$$w_{t,d} = \mathrm{tf}_t \cdot \log \frac{|D|}{|\{t \in d\}|} \tag{1}$$

In the expression, $w_{t,d}$ is the weight of term t in document d; tf_t is the term frequency of term t in document d; $|D|$ is the total number of documents; $\log \frac{|D|}{|\{t \in d\}|}$ is the inverse document frequency; $|\{t \in d\}|$ is the total number of documents containing the term t.

Suppose $1, 2, .., N$ denote the terms of a document d. Then the weighted document vector $\mathbf{v}_d$ for d is:

$$\mathbf{v}_d = [w_{1,d}, w_{2,d}, \ldots, w_{N,d}] \tag{2}$$

We use cosine similarity to measure the degree of "semantic closeness" between the two vectors. Given a query vector v_q, the cosine of the angle between this vector and a document vector d is the expression:

$$\cos\theta = \frac{\mathbf{v_q} \cdot \mathbf{v_d}}{\|\mathbf{v_q}\| \, \|\mathbf{v_d}\|} \tag{3}$$

A value of 1 means the vectors are identical, and it is 0 if they are orthogonal. Two vectors are considered to be *similar* if their cosine similarity is above some set threshold.

We introduce the notion of an *Instruction-Class*, which is a class of similar instructions in test scripts and members of which perform similar functions across tests (*e.g.*, entering a password into a textbox in a login form). We map each instruction i_j in a script to an Instruction-Class $IC(i_j)$. For example, the instruction *enter "12345" into the second textbox* is different from the instruction *enter "xyzabc" into the "Password" textbox*. However, both of them indicate entering password into a textbox. Since they are similar, we would like them to be mapped to the same Instruction-Class. For simplicity, we will denote $IC(i_j)$ as l_j throughout the paper.

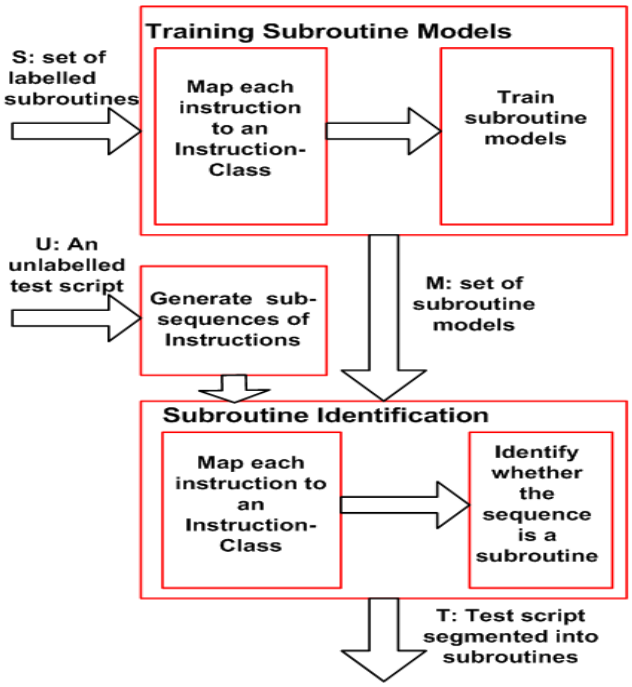

Figure 7. High level overview of the algorithm

Overview of the Proposed Approach

The goal of subroutine identification is to automatically identify subroutines contained within a repository of scripts. We assume that a user has previously labeled several instances of each desired subroutine within a separate training repository. Let S denote the set of such labelled subroutines. The goal of the algorithm is to recognize instances of these subroutines within the unlabeled scripts in the main repository. Figure 7 illustrates the high level overview of our algorithm.

Each labeled subroutine consists of a sequence of instructions. Our algorithm works as follows. First, each instruction is mapped to an Instruction-Class. Second, sequences of Instruction-Classes which are labeled as being instances of the same subroutine are used to train the subroutine model. The output of this step is the set of subroutine models, M. Finally, given an unlabeled script U, which may contain subroutines, all possible subsequences of instructions in that script are examined and compared against each of the subroutine models. If a match is found, the algorithm concludes that the matching instruction sequence is an instance of the matching subroutine. The output of this process is the script T, which is segmented into subroutines. The next subsections describe each of the components of this algorithm in detail.

Mapping Instructions to Instruction-Classes

Given a set of instructions I = $\{i_1, i_2, \ldots.\}$ and a set of Instruction-Classes IC = $\{l_1, l_2, \ldots.\}$, each instruction i_j is mapped to the Instruction-Class in IC to which it is most similar.

This is a clustering problem where each Instruction-Class is

a cluster. Such clusters are constructed from instructions in scripts. Features of the instruction are action type, object type, words and word combinations (unigrams, bigrams, trigrams) from object label. We represent these instructions as vectors. Features of the instruction become the terms of the vector representing that instruction.

We use the parser described by Lau et al. [14] to parse the instructions and identify the type of action, type of the object, object label and value. For example, the instruction *click the "add to cart" button* is parsed into the following information:

- Action Type: click
- Object Type: button
- Object Label: "add to cart"

The features of this instruction are the following: (click, button, "add", "cart", "add to", "to cart", "add to cart"). Here, words and word combinations (bigrams, trigrams) are computed from the object label. These features become the terms of the vector representing the instruction. The vector which represents this instruction is: <click, button, "add", "cart", "add to", "to cart", "add to cart">. These terms are weighted using a TF*IDF weighting scheme. If the action type of an instruction is "assert", we do not add the "assert" keyword to the vector representing that instruction. This is to ensure that an assertion can be considered to be similar to any other instruction that acts on similar objects. Thus, the vector which represents *assert there is a "add to cart" button* is: <button, "add", "cart", "add to", "to cart", "add to cart">.

To compute similarity of an instruction to an Instruction-Class, we do the following:

- If the action type of the instruction is not "assert" and is different from the action type of the Instruction-Class, then the instruction is not similar to the Instruction-Class. Thus, the instruction *click the "username" textbox* is not similar to the instruction *enter name into the "username" textbox*.
- Otherwise, we compute a cosine similarity score between the vectors computed from the instruction and the Instruction-Class. If this is above the cosine similarity threshold of clustering (determined experimentally), then the instruction is considered to be similar to the Instruction-Class.

Each instruction is assigned to the Instruction-Class to which it is most similar. When an instruction is assigned to an Instruction-Class, we update the terms of the corresponding vector with the terms of the instruction and adjust the TF*IDF score of the Instruction-Class vectors. However, it is possible that an instruction may not be assigned to any Instruction-Class (i.e. the cosine similarity is below the threshold for every Instruction-Class or the set of Instruction-Classes is empty). In that case, we create a new Instruction-Class from that instruction.

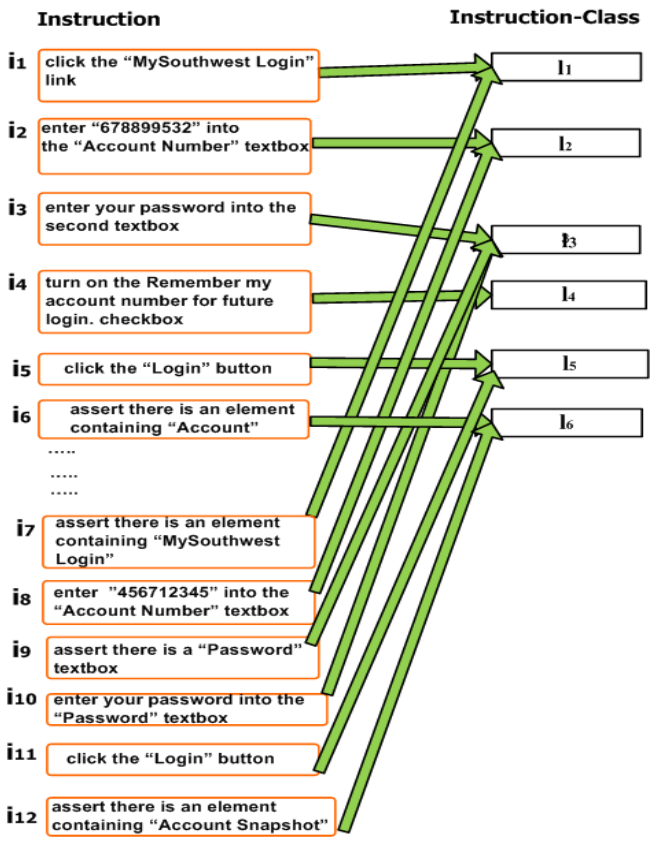

Figure 8. Mapping instructions to Instruction-Classes

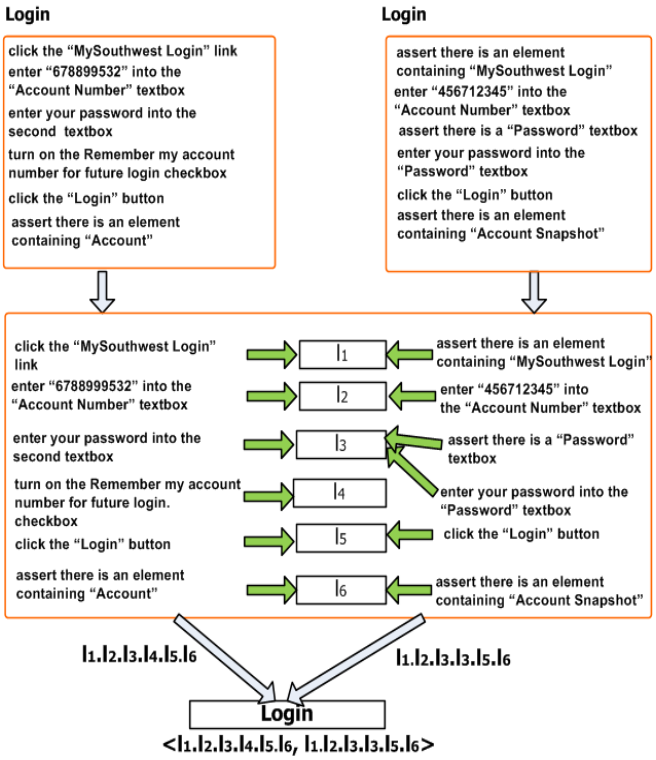

Figure 9. Training "Login" subroutine vector

Let us illustrate the algorithm with the instructions for the "Login" subroutine as illustrated in Figure 2 (a) and (b). Initially the set of Instruction-Classes is empty. Therefore, we create an Instruction-Class vector from the terms of the first instruction i_1 and assign a machine-generated identifier (l_1) to that Instruction-Class. The term vector of the next instruction i_2 is compared with the term vector of this Instruction-Class. But the similarity value is below the threshold. As a result we create another Instruction-Class vector from the terms of that instruction and assign it an identifier (l_2). Similarly, Instruction-Class vectors are created from the next four instructions and assigned machine generated identifiers (l_3, l_4, l_5, l_6).

The first instruction of the "Login" subroutine in Figure 2 (b) is found to be similar to the first Instruction-Class. As a result, we add the terms of that instruction to the Instruction-Class vector. The next instruction i_8 is found to be similar to the second instruction class l_2, and the terms of this instruction is added to the Instruction-Class vector. The next two instructions are found to be similar to the third Instruction-Class and the final two instructions are found to be similar to the fifth and sixth Instruction-Classes. Figure 8 shows the Instruction-Class labels constructed from these instructions.

Training Subroutine Models

Training a subroutine model[1] consists of constructing a vector for that subroutine from the labelled instances collected from test scripts. For each subroutine instance, we identify the Instruction-Class from each of the instructions. We construct the terms of the subroutine vector from the resulting Instruction-Class sequences. Given a sequence of Instruction-Classes $l_1.l_2.l_3$ labelled as an instance of subroutine S, the term of the subroutine vector V_S is the sequence $l_1.l_2.l_3$. Note that terms are typically words and word combinations in a vector space model. However, in our representation, each term of the subroutine vector is a sequence of Instruction-Class labels.

Figure 9 shows how a subroutine vector is constructed from labelled instances of the "Login" subroutine collected from the scripts shown in Figure 1. Two labelled instances of the "Login" subroutine are given as examples. The instructions from each subroutine are mapped to Instruction-Classes. As a result, an Instruction-Class sequence is retrieved from each subroutine instance. For example, the sequence $l_1.l_2.l_3.l_4.l_5.l_6$ is retrieved for the first subroutine. This sequence becomes a term of the "Login" subroutine vector. Similarly, the sequence $l_1.l_2.l_3.l_3.l_5.l_6$ becomes another term of the "Login" subroutine vector. We weight each such term using a standard TF*IDF weighting scheme (which computes a weight using the frequency of this sequence in this particular subroutine vector and all the other subroutine vectors). For lack of space, Figure 9 does not show the other subroutine vectors (*e.g.*, a "Checkout" subroutine vector) constructed from the scripts recorded from that website.

[1] In this paper, we use the terms subroutine vector and subroutine model interchangeably.

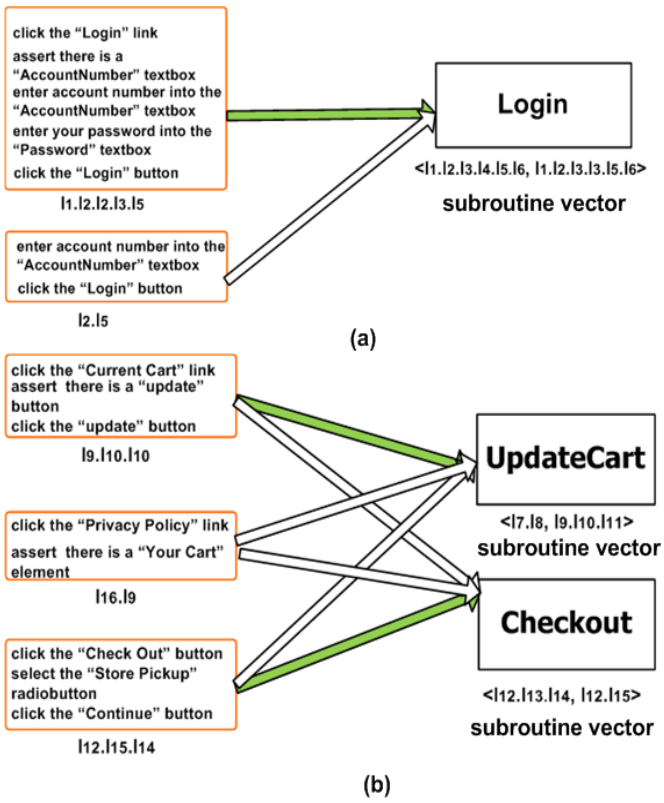

Figure 10. Subroutine Identification

Subroutine Identification

Trained subroutine models are then used to identify unlaballed instances of subroutines. To determine whether a sequence of instructions in a test script is an instance of a learned subroutine, we do the following:

First, we identify the Instruction-Class for each of the instructions. Next, we construct a term from the resulting Instruction-Class sequence and build a vector with that term. Then, we compute the cosine similarity of this vector with each of the trained subroutine vectors. If the similarity score between them is above the cosine similarity threshold for subroutine identification (determined experimentally), the sequence of instructions is identified as an instance of that subroutine. If the sequence is identified as an instance of multiple subroutines (i.e. cosine similarity is above the thresholds for multiple subroutine vectors), then the highest scoring vector is picked as the final classification.

However, we modify the definition of cosine similarity in order to capture the variability of instruction sequences of a subroutine. The usual cosine similarity measure considers two terms in a vector to be equal iff they are exactly identical. We extend this and consider two terms to be equal iff any of the following conditions are satisfied:

- The terms are identical.
- One of the terms is a generalization of the other term.

- One of the terms is a partial match of the other term.

To compute whether a term (i.e. the Instruction-Class sequence) is a generalization of another term, we remove any repeated subsequence from it, and see whether this is identical to the other one. For example, removing the repeated occurances of l_4 from the term $l_1.l_2.l_3.l_4.l_4.l_5$ makes it identical to the term $l_1.l_2.l_3.l_4.l_5$.

To compute whether a term (i.e. the Instruction-Class sequence) partially matches another term, we compute the edit distance [11] between them, normalize the edit distance by the length of the larger term and conclude a partial match if the normalized distance is below the threshold set for edit distance (determined experimentally). For example, the edit distance between the terms $l_1.l_2.l_2.l_3.l_5$ and $l_1.l_2.l_3.l_4.l_5$ is 2. The normalized value is 0.4 which is below the edit distance threshold 0.5. Hence this is a partial match.

Figure 10 (a) and (b) illustrate subroutine identification. For Figure 10 (a), the first sequence of instructions is identified as an instance of the "Login" subroutine. In Figure 10 (b), the first sequence is identified as "UpdateCart", the third sequnence is identified as "Checkout" and the second sequence is not identified as any of the subroutines.

Segmenting a Script into Subroutines

Given a test script as input, our algorithm can determine the subroutines from it and segment the entire script into subroutines. It generates consecutive subsequences of instructions from the script in descending order of the size and identifies whether the subsequences are subroutines. Algorithm *SegmentScript* illustrates the high-level abstract pseudo-code.

Algorithm *SegmentScript*
Input: $Script$: A Test Script
Input: $Models$: Trained Models of Subroutines
Output: *ScriptWithSubroutines*: Script which has Subroutines identified

```
1.  Instructions ←Instructions of Script
2.  ScriptWithSubroutines ←Script
3.  for i ← 1 to Instructions.Length
4.      do for j ← Instructions.Length downto i
5.          do CurSeq ←Sequence of Instructions from
               i to j
6.             IdentifySubroutine (CurSeq, Models)
7.             if CurSeq is a Subroutine
8.               then Label CurSeq as Subroutine
9.                    Add this Label to Script
10.                   i ←j + 1
11. return ScriptWithSubroutines
```

Algorithm *SegmentScript* starts with the largest subsequence, i.e. the entire script and checks whether this is a subroutine. In case of a match, it labels the subsequence as a subroutine and does not check for more subroutines inside it. Otherwise, it generates subsequences of length n - 1, and checks for subroutines. If a subroutine is found, it labels the subsequence with the subroutine name and checks for more subroutines outside the subsequence. Otherwise, the algorithm

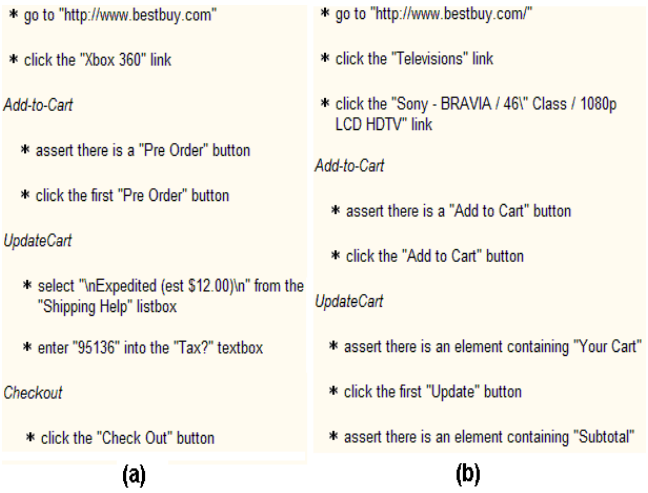

Figure 11. Test scripts segmented into subroutines

generates smaller subsequences (n - 2, n - 3, ..., 2, 1) and checks for subroutines.

For example, Figure 11 shows a test script segmented into subroutines.

EVALUATION

We present the experiments we have done to evaluate the subroutine identification feature of CoTester and a preliminary lab study that illustrates the value of the system.

Dataset

We used 70 scripts recorded from 12 websites for our experiments. They were recorded by active users of the CoScripter system [16]. We manually identified 144 subroutines from the scripts. Table 2 shows the experimental dataset. The first column of this table shows the websites, the second column shows the total number of scripts (in our dataset) which were recorded from that website, the third column shows the number of subroutines manually identified from those scripts, the fourth column shows the training set, and the final column shows the testing set. Each comma-seperated item in these sets specifies a subroutine and its number of instances in the set.

Baseline Algorithm

We wanted to justify our use of Instruction-Class sequences as terms of the subroutine vector. To do that, we also implemented a simpler baseline algorithm, Subroutine-Identification-Simple, which ignored the sequences, constructed subroutine models using the bag of words from the instructions, and used only the bag of words from instruction sequences to identify subroutines.

Performance

Website testers usually create a test suite from scripts recorded from a single website. Therefore, we assessed the performance of our algorithm by constructing website-specific subroutine models. Subroutine identification from scripts

Website	Scripts	Subroutines	Training Set	Testing Set
Amazon	7	16	Login:2, Add-to-Cart:2, Checkout:2, Search:1	Login:3, Add-to-Cart:2, Checkout:2, Search:2
BN	5	13	Login:2, Search:2, Add-to-Cart:1,Continue:2	Login:2, Search:2, Add-to-Cart:1, Continue:1
OfficeMax	6	15	Search:2, Add-to-Cart:2, Checkout:2, Register:1	Search:2, Add-to-Cart:2, Checkout:2, Regiser:2
Typetees	8	19	Login:3, Search:2, Register:2, Continue:2, Logout:1	Login:2, Search:2, Register: 1, Continue:2, Logout:2
Walmart	7	11	Search:2, Continue:3, Checkout:2	Search:2, Continue:1, Checkout:1
OfficeDepot	5	11	Login:2, Logout:2, Register:1	Login:2, Logout:2, Register:2
CircuitCity	5	10	Add-to-Cart:2, Continue:2, Checkout:1	Add-to-Cart:2, Continue:2, Checkout:1
Bestbuy	4	8	Checkout:2, Register:2	Checkout:2, Register:2
Shop	6	12	Checkout:2, Logout:2, Login:2	Checkout:2, Logout:2, Login:2
Buy	7	10	Login:2, Add-to-Cart:2, Search:2	Login:2, Add-to-Cart:1, Search:1
Threadless	4	9	Add-to-Cart:2, Register:1, Checkout:1	Add-to-Cart:2, Register:2, Checkout:1
Theselectseries	6	10	Search:2, Register:3	Search:2, Register:3

Table 2. Experimental Dataset

across multiple websites is more challenging since instructions in the scripts may show a lot of variability across websites.

We constructed such subroutine models for each website in our dataset and tested their performance. We computed how many subroutines were identified correctly, how many were identified incorrectly, how many were not identified. From this computation, we measured recall/precision and F-measure of the learned subroutine models. We averaged these values across websites. On average, our algorithm achieved 94% precision, 89% recall and 91.8% F-measure for subroutine identification.

We compared this performance with that of our baseline algorithm. We found that F-measure performance was 14% lower for this simpler approach. (see Figure 12, overall F-measure for the simpler approach is 77.8%). The performance differences are statistically significant (two-tailed p value is less than 0.0001, 95% confidence, t = 6.4920, degrees of freedom = 10, standard error of difference = 0.021). This justifies our use of sequences to identify subroutines.

We were interested to find whether the use of cosine similarity instead of simple equality checking between an instruction and instruction-class was really needed. Since cosine similarity with a threshold of 1.0 is equivalent to equality checking, we varied the cosine similarity threshold of clustering from 0 to 1.0 by an increment of 0.1 and computed the performance. A cosine similarity threshold of 0.4 for clustering resulted in the highest performance when the other two thresholds (edit distance threshold and cosine similarity threshold of subroutine identification) were set to 0.5 (the performance reported in this paper is for these threshold values). Performance drops by 12% when the cosine similarity threshold of clustering was set to 1.0 (which is equality checking), when the other thresholds were 0.5. This performance drop is statistically significant (two-tailed p value is less than 0.0001, 95% confidence, t = 6.3095, degrees of freedom = 12, standard error of difference = 0.018). This justifies the use of cosine similarity instead of simple equality checking between an instruction and the Instruction-Class.

User Study

We also did a preliminary user study of the system. The goal of this study was to find whether the easy-to-understand scripting language and subroutine identification features of CoTester would be useful to users. 4 users participated in the study. They were experienced computer users and regularly browse the web. 3 of them had previous web development experience. 2 users were familiar with web testing tools and 1 user previously used a web testing tool. We showed them the user interface of CoTester and some example test scripts. We also introduced them to the subroutine learning feature. Each participant recorded 2 scripts from an e-commerce website (a total of 4 websites were used in the study, 1 website per participant) such that each script contained at least the following subroutines: "Login", "Add-to-Cart", "Checkout". They also added assertions to each of the scripts in two ways: i) using the graphical user interface ii) manually editing the scripts. Participants mentioned that they would prefer the first approach to insert assertions. Then, we used our algorithm to identify the subroutines from the scripts and showed the modified scripts to the participants. They noted that subroutines segmented the scripts in conceptual units, and helped them to better understand the higher level tasks performed by instructions. They also ran the scripts in batch mode and saved the test reports. One of the participants mentioned that our test report should contain information about the subroutine when a failure occured within that subroutine (*e.g.*, the "Login" button is not found). This can be a novel benefit of subroutines we did not anticipate. Finally, we asked them to edit the recorded scripts. They noted that although they had to look at the repository of other scripts to get the correct syntax of instructions, editing was quite easy since the scripts were understandable. Overall, all of them liked the simple language of the CoTester system. One of the participants mentioned that she would like to write test scripts using our system to test her personal website.

CONCLUSIONS AND FUTURE WORK

We have presented CoTester, a lightweight web testing tool which can help testers easily create and maintain test scripts. CoTester's easy-to-understand scripting language and subroutine identification feature can reduce, if not eliminate, the barrier to web application testing.

There are many possible avenues of future research: First, CoTester's assertions check for presence or absence of web page elements based on textual properties (*e.g.*, caption of a button). In the future, we would like to add other forms of

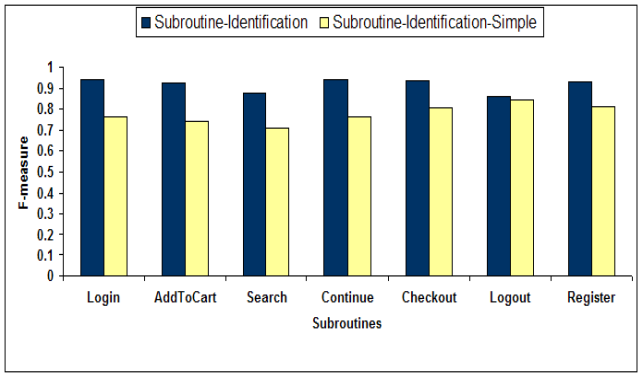

Figure 12. Subroutine Identification Performance

assertions, *e.g.*, an assertion that can check whether a submit button is disabled, or an assertion that can perform a bitmap comparison. Second, the subroutine identification feature is not yet deployed to functional testers. We like to deploy this feature to functional testers and see whether this can improve test script maintenance by enabling them to automatically apply similar changes to a large corpus of scripts (bulk modification), when websites change. Towards that, we will extend the user interface of CoTester with a bulk modification component, so that testers can select the scripts or the test suite and specify the desired change (e.g., adding an assertion in a login process) and the system can do the modification in all the matching scripts. Third, we have trained and tested subroutine models for a few subroutines. In the future, we will train more subroutine models, conduct thorough experiments with a larger dataset, and explore the usage of subroutine models trained from multiple websites. Finally, we will perform a thorough user study of the deployed system.

ACKNOWLEDGEMENT

We thank Jeffrey Nichols for his insightful comments about this paper.

REFERENCES

1. Borland SilkTest. http://www.borland.com/us/products/silk/silktest/.
2. Hp QuickTest Professional. http://www.hp.com.
3. IBM Rational Functional Tester. http://www-01.ibm.com/software/awdtools/tester/functional/.
4. J. F. Allen, N. Chambers, G. Ferguson, L. Galescu, H. Jung, M. D. Swift, and W. Taysom. Plow: A Collaborative Task Learning Agent. In *Proc. of AAAI*, 2007.
5. J. Blythe. Task Learning by Instruction in Tailor. In *Proc. of the 10th Intl. Conf. on Intelligent user interfaces*, pages 191–198, New York, NY, USA, 2005. ACM.
6. M. H. Burstein, R. Laddaga, D. McDonald, M. T. Cox, B. Benyo, P. Robertson, T. Hussain, M. Brinn, and D. V. McDermott. Poirot - Integrated Learning of Web Service Procedures. In D. Fox and C. P. Gomes, editors, *AAAI*, pages 1274–1279. AAAI Press, 2008.
7. `http://www.concordion.org/`.
8. A. Cypher. Watch What I Do: Programming by Demonstration. *MIT Press*, 1993.
9. `http://www.w3.org/DOM/DOMTR`.
10. M. Gervasio, T. J. Lee, and S. Eker. Learning Email Procedures for the Desktop. In *AAAI 2008 Workshop on Enhanced Messaging*, Chicago, IL, July 2008.
11. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, January 1997.
12. S. B. Huffman and J. E. Laird. Flexibly Instructable Agents. Technical report, Price Waterhouse, 1995.
13. T. Lau, L. Bergman, V. Castelli, and D. Oblinger. Sheepdog: Learning Procedures for Technical Support. In *Proc. of Intl. Conf. on Intelligent user interfaces*, pages 109–116, 2004.
14. T. Lau, C. Drews, and J. Nichols. Interpreting Written How-to Instructions. In *Proceedings of the International joint conference on Artificial Intelligence*, 2009.
15. T. A. Lau and D. S. Weld. Programming by Demonstration: An Inductive Learning Formulation. In *Proc. of Intl. Conf. on Intelligent User Interfaces*, pages 145–152, 1999.
16. G. Leshed, E. M. Haber, T. Matthews, and T. Lau. Coscripter: automating and sharing how-to knowledge in the enterprise. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 1719–1728, 2008.
17. G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 943–946, 2007.
18. D. S. Rosenblum. A Practical Approach to Programming With Assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995.
19. `http://sahi.co.in/w/`.
20. G. Salton, A. Wong, and C. S. Yang. A Vector Space Model for Automatic Indexing. *Commun. ACM*, 18(11):613–620, 1975.
21. `http://seleniumhq.org/`.
22. A. Spaulding, J. Blythe, W. Haines, and M. Gervasio. From Geek to Sleek: Integrating Task Learning Tools to Support End Users in Real-world Applications. In *IUI '09: Proceedings of the 13th international conference on Intelligent user interfaces*, pages 389–394, 2009.