Using Assertions to Help End-User Programmers Create Dependable Web Macros

Andhy Koesnandar†, Sebastian Elbaum†, Gregg Rothermel†,
Lorin Hochstein†, Kathryn Thomasset†, and Christopher Scaffidi*

†Computer Science Department
University of Nebraska – Lincoln
Lincoln, NE, U.S.A.
{akoesnan, elbaum, grother, hochstein, kthomass}@cse.unl.edu

*School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, U.S.A.
{cscaffid}@cs.cmu.edu

ABSTRACT

Web macros give web browser users ways to "program" tedious tasks, allowing those tasks to be repeated more quickly and reliably than when performed by hand. Web macros face dependability problems of their own, however: changes in websites or failure on the part of end-user programmers to anticipate possible macro behaviors can cause macros to act incorrectly, often in ways that are difficult to detect. We would like to provide at least some of the benefits of software engineering methodologies to the creators of web macros. To do this we adapt assertions to web-macro programming scenarios. While assertions are well-known to professional software engineers, our web macro assertions are unique in their focus on website evolution, are generated automatically, and encode the expectations and assumptions of a rapidly growing group of users who often have limited formal programming expertise. We have integrated our techniques for assertion generation and evaluation into a web macro tool, and performed an empirical study investigating its use. Our results show that the assertions can help web macro users detect macro failures and correct macro faults.

1. INTRODUCTION

Web macros give web browser users ways to "program" for the web, saving time and reducing the incidence of errors resulting from manual repetition of tasks. Web macros are being used in a wide range of practical situations to help users perform important jobs; for example, analysts at *Home2Hotel.com* use web macros to manage online advertising and compile sales information [9], auditors at the World Bank use web macros to perform automatic quality audits on their intranet sites [8], and product managers at Audi use web macros to perform competitive analyses of automotive lines [10]. Web browsers are beginning to connect with the potential for web macros as well, incorporating architectural plug-in facilities designed to help users program and use them.

To create web macros, users employ web macro tools, most of which utilize a programming-by-demonstration (PBD) approach [4] to encode actions. Under the PBD paradigm, users manually demonstrate a sequence of actions involving webpage access and manipulation (e.g., navigating through a webpage and filling out forms), and the tool records a generalized description of these actions. PBD-based web macro tools have a broad user appeal because they do not require expertise in writing code.

While web macros can be enormously helpful, they can also cause several dependability problems [17, 21]. For example, when changes have been made to the websites on which a macro operates, that macro may terminate prematurely, produce incorrect results, or perform unintended actions [21]. Other types of errors occur when a user constructing a macro fails to anticipate potential responses to that macro's operation (as when a resource assumed by the macro to always exist is not found), or when a user expects potential types of data to be encountered (e.g., names entered family-name-first rather than given-name-first). The resulting errors can be difficult to detect, leading to unwarranted confidence in the results of macros. Since the tasks automated by web macros can be critical to business or other processes and can have costly side-effects if performed incorrectly [9, 21], finding ways to help the creators of macros detect and correct such problems is important.

We would like to provide at least some of the benefits of software engineering methodologies to the creators of web macros. We have therefore developed an approach for improving web macro dependability by encoding users' expectations and assumptions about macros into assertions. These assertions are automatically generated as macros are created, and then, during subsequent web macro execution, results of the assertions are used to detect unintended consequences. To evaluate our approach, we implemented Robofox, a Mozilla Firefox-based web macro tool, which incorporates our techniques. Our empirical study of users employing Robofox show that the approach helps them detect macro failures, isolate the associated faults, and correct their macros.

The notion of using software engineering methodologies to help end users program more dependably in non-traditional language paradigms is not new in the software engineering research literature; in particular, it has been employed successfully in relation to the spreadsheet language paradigm (e.g., [6, 19]). The notion of using assertions to enhance software dependability is also not new in the context of software engineering (e.g., [12, 18]), and prior research has also considered automatic assertion generation in the context of professional programming [14]. These lines of thought have come together in work utilizing range assertions to help end-user programmers create dependable spreadsheets [2].

This prior work notwithstanding, addressing dependability needs for end users working in the web macro paradigm presents new challenges. The first challenge involves the web macro environ-

ment, which is quite different from programming contexts considered in prior work (whether spreadsheets or traditional programming languages). Web macros live in a harsh climate: clipped regions of a web page can appear and disappear as a site evolves, form fields can be added and removed, and variables associated with those fields do not even have types. Consequently, things that can be taken for granted in other programming contexts must be more carefully checked in web macro environments, motivating the need for assertions.

A second challenge involves the characteristics of the end users who program web macros. While web macro programmers may be fairly skilled in their jobs (as we would expect the analysts, auditors, and product managers referred to above to be), they also tend to have little interest in programming for its own sake; if they program it is in order to perform some task important to their "real" jobs. We find a wide range of skills among end users who program web macros [20]: most could not be expected to write or attend to the details of assertions, but some do have a great deal of expertise. Our primary focus in this work is on the former class of users, but we do provide some support for the latter as well.

The research we report here is novel along several dimensions:

- This is the first research to attempt to bring software engineering methodologies (and in particular the use of assertions) to play within the web macro programming paradigm, to help end users create and use these macros dependably.
- The family of assertion types we utilize, which focus on particular website aspects such as the presence of specific design elements, the flow and format of data passed between websites and associated applications, and the state of a given website when a request is made, are novel.
- The underlying mechanisms we use to support the generation, checking, and handling of such assertions, which must be hidden within the web browsing environment while still allowing for the evolution of the macros, have not previously been investigated.

The rest of this paper is organized as follows. Section 2 provides background information on web macros and their use, web macro tools, and the relation of these tools to other engineering support devices such as testing tools. Section 3 describes our automated mechanisms for generating and evaluating assertions. Section 4 presents our study of the use of assertions in web macros. Finally, Section 5 presents conclusions and discusses possible future work.

2. BACKGROUND

We now describe two scenarios involving web macros that we use in the remainder of this paper to illustrate our approach. We then describe the web macro tool that we have implemented to support the work, and its relation to existing tools.

2.1 Web Macro Scenarios and Faults

Insurance Quotes. Consider an independent insurance agent who wishes to find the best insurance quote for a customer. When a customer makes an inquiry, the agent must query multiple insurance companies or insurance comparison websites. Since these websites require many pieces of data about the customer (e.g., name, address, age, weight, occupation, coverage required, tobacco usage, family history, insurance history), the agent saves the customer's information in a spreadsheet from which it can be copied and pasted into the sites' forms. The agent repeats this process across multiple sites and provides the best quote to the customer.

A web macro can automate this process by populating insurance quote sites with data from the corresponding spreadsheet cells and making the data-entry process more efficient and less error prone. However, other types of failures can occur with such a macro: for example, when an insurance site changes its quotes display from a list to a tabular format, a macro may be unable to clip the desired information. When an insurance site changes the date format, the macro may work but return the wrong quote. When a site alters its default values, the macro will eventually provide the wrong quote. Consider the site *insure.com* for example, which in December 2006 changed the default value in the coverage amount field from $50K to $500K. A macro using this site before December 2006 would provide insurance quotes for coverage at the $50K level. After that date, the same macro would provide a higher level of coverage with a corresponding higher price. An insurance agent unaware of the change would give a quote with the wrong coverage level.

Data Migration. Consider a data entry task aimed at incorporating student data from a university's database into a bookstore's customer database. A bookstore may use the student information to send out promotions and booklets each semester. For students at the University of Nebraska - Lincoln (UNL), this task can be performed through UNL's "PeopleFinder" website, using copy-and-paste actions to place data into the bookstore's internal contact database web interface. To do this, for each student, a clerk must perform the copy-and-paste operation for the student's name and address.

At UNL this data migration task could entail 20,000 copy-paste operations, which is highly repetitive and error prone. Still, the use of a web macro to automate this task could lead to other types of failures. The student information in the Peoplefinder website may vary depending on the students' place of residence and the information they provide. If a student lives on-campus, Peoplefinder's website displays their address with a nine-digit zip code, but if they live off-campus, the website displays their address with just a five-digit zip code. Furthermore, some students may not provide their local address information to the university. A web macro built with the assumption that all students' information includes nine-digit zip codes will fail to extract the zip codes for students who live off-campus or fail to provide addresses.

The foregoing scenarios illustrate some of the types of tasks end users can create web macros to perform. While we might deplore a decision to entrust such an important task as, say, underwriting to a brittle program like a web macro, the instances of web macro usage cited in Section 1 indicate that users currently *do* use web macros to perform tasks just as important. Further, as illustrated in both scenarios, end users creating web macros do not control and often do not fully understand the potential behavior of the websites accessed by their macros. As such, the web macros they create are extremely vulnerable to changes in a website's structure, semantics, or default behavior, and also likely to underapproximate a website's exceptional behavior.

2.2 The Robofox Web Macro Tool

There are many tools that enable recording and replay of user actions on web clients. Several tools have been developed, for example, to support software testing activities by providing an infrastructure that helps automate test case execution [15, 16]. Web macro tools such as Newbie Web Automation[13], Deskperience Web Replay [5], Kapowtech [10], and CoScripter [1, 3], also provide recording and replay facilities that replicate user interactions with websites through the browser, though with less concrete connection to testing.

Although the lines between web testing tools and web macro tools are often blurred (e.g., iOpus iMacros [8] is said to support testers), one distinguishing aspect of web macro tools is that they

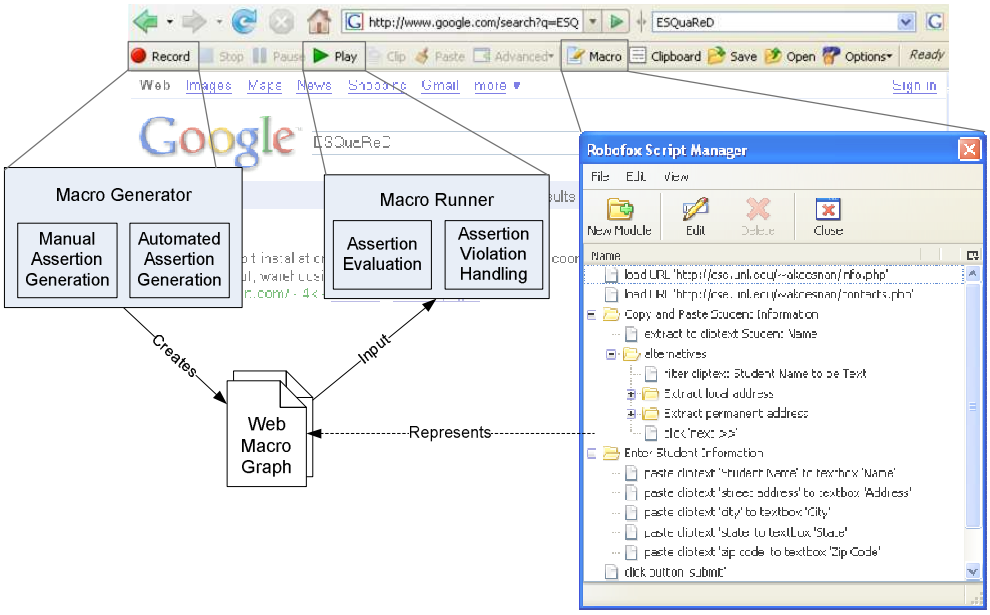

Figure 1: Robofox's interface and high level architecture.

are built not only to operate on a user's own websites, but also to manipulate websites that are not under that user's control, and which may be subject to unannounced changes. To cope with this situation, web macro tools tend to incorporate more sophisticated mechanisms to identify website elements even in the presence of changes, built-in facilities to edit the macros within the browser while browsing, and parameterized macros that can be shared across a user community [3]. Still, none of the testing or web macro tools just discussed incorporates any notion of assertions, and all of them can run wild when a target web site changes.

In this work we use Robofox, a web macro tool that we have implemented, to study our automated assertion generation and checking techniques. Robofox does not require users to have programming experience, focusing instead on allowing users to work as they normally do within the context of a web browser. We have integrated Robofox into the Firefox browser as a toolbar extension that provides access to functionality with which users record and replay macros. Figure 1 shows the Robofox toolbar in the browser, its script manager, and its basic architecture. From the users's perspective, once recorded, a macro is displayed in the Script Manager as a tree of action-object pairs (e.g., action "load" and object "http://cse.unl...", action "click" and object "submit button").

In terms of generality, the architecture of Robofox is similar to that of other PBD-based web macros [11, 21], and the set of features Robofox provides include the primary features available in commercial tools (e.g., the abilities to clip website content, format clipped data, paste data from multiple clipboards, import and export data from a file or a spreadsheet, and obtain notifications when a website changes).[1] A user creates a macro with Robofox by initiating a recording session and then interacting as usual with websites of interest via the web browser. Robofox's Macro Generator component captures the user's interactions and generates the web macro. For each user interaction, the Macro Generator captures event information and processes it to generate an action-object pair used later to replicate the interaction during replay. Table 1 lists several actions that Robofox captures. To replicate interactions that operate on objects on webpages, Robofox identifies these objects by recording their XPaths, names, and ids, and visual patterns that help to pinpoint an object (e.g., a button's neighboring labels [1]).

Table 1: A Sample of Actions Captured by Robofox.

Class	Action	Description
Navigation	Click	Click element
	Go Back	Go to previous page in browser history
	Go Forward	Go to next page in browser history
	Load	Load a URL
Form Manipulation	Check	Check checkbox
	Choose	Choose radio button
	Enter	Enter value into textbox
	Paste	Paste clipboard value to form fi eld
	Select	Select option from dropdown
	Uncheck	Uncheck checkbox
Data	Clip	Place text/HTML element on clipboard
	Defi ne	Defi ne new clipboard
	Delete	Delete existing clipboard
	Export	Export clipboard values
	Filter	Extract text matching pattern
	Import	Import clipboard values
Other	Close Tab	Close browser tab
	Notify	Pop-up message box
	Verify Exist	Verify text present on page
	Verify !Exist	Verify text absent from page
	Wait	Wait for page refresh

Internally, Robofox represents a web macro as a Web Macro Graph (WMG). A WMG is a directed graph $G = (V, E)$, much like the control flow graphs used to represent code written in an imperative language, where V is a set of nodes and E is a set of directed edges. A node $v \in$V can represent the entry to (initiation of) a macro, an action captured in the recording session, a branch or merge in the flow of actions captured, or a macro exit. Edge direction allows Robofox to enforce the order of execution.

Continuing with Figure 1, Robofox's Macro Runner provides the environment for replicating the interactions recorded in a macro. During a replay session, Robofox reads the WMG associated with a previously recorded macro to reproduce the encoded user interactions. When an action node is visited, Robofox attempts to find the target element based on its XPath, element name, element id,

[1]Our decision to implement Robofox was a practical one, given that source code for existing web macro tools was not available to the extent necessary to allow us to explore our techniques to improve web macro dependability.

or a visual pattern. If the element is found, the action is executed on that element; otherwise Robofox displays an alert and the macro execution is stopped. If the action completes successfullly, Robofox moves to the next node, otherwise it stops execution. When Robofox finds a WMG branching node, it evaluates its associated predicate to determine the next node to visit. A traversal of the WMG is complete when the exit node is reached.

The Macro Generator and Runner also contain the assertion generation, evaluation, and handling mechanisms that are the novel contribution of this work; these are discussed next.

3. ASSERTIONS IN WEB MACROS

We now describe the primary contribution of this work: our mechanisms for generating and evaluating assertions in web macros.

3.1 Assertion Generation

As we have noted, some web macro programmers may reach a high-level of expertise in dealing with macros, and these users might be able to "manually" construct certain types of assertions and insert these into web macros at appropriate points. Robofox supports one such manual process, similar to existing commercial web testing tools [15], by enabling the user to specify visual cues on webpages that correspond to the appearance of certain elements. Users can generate such assertions by simply selecting text on the webpage and clicking the "verify exist" context menu, as pictured in Figure 2. These assertions become part of the macro script and are checked by Robofox in the background at run-time (details on these later steps are provided in Section 3.2).

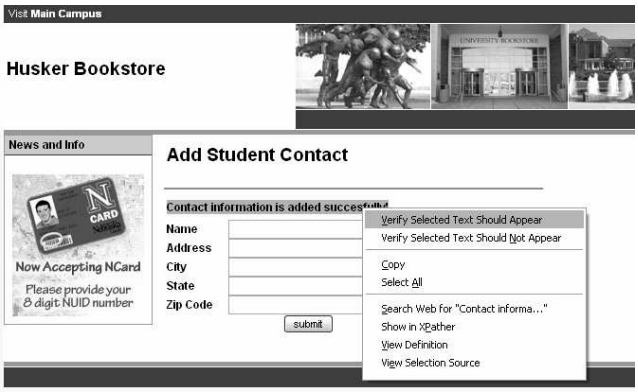

Figure 2: Manual assertions are entered via context menus that allow users to verify whether selected text should appear or not.

For most web macro programmers, however, automated assertion generation techniques are more viable, so we focus on these here. Automated techniques require only limited user participation while providing powerful controls for the detection of subtle changes in macro operation that may have a large impact on macro dependability. We concentrate on three classes of assertions:

- *Existence assertions* verify whether elements and data available during the recording session are available in the replay session. We define an HTML object (e.g., text, bullet, figure, checkbox) to be available if it can be located by the web macro tool. We define a clipboard variable as available if it is defined and its value is not null. *Non-existence* assertions, the counterpart of existence assertions, are used to verify the absence of a webpage object and to detect collisions caused by attempts to rewrite an existing clipboard variable.
- *Value assertions* verify that the value of a clipboard variable or HTML object seen during macro replay matches a specified value or value characterization (e.g., is not empty, is greater than) from a recorded execution. For most HTML objects we derive the value from the text corresponding to the object (e.g., text found in a label, next to a bullet, or in a specific table row). For a form's elements, we define the value through the corresponding HTML "value" construct (e.g., "checked" for checkboxes, "text" for text input fields).
- *Data type assertions* verify whether the format of the data used by an action is appropriate. In Robofox, clipboard data consists of text strings. We enforce the type of such data by specifying or inferring its format. For example, a clipboard variable can be declared to have a "zip code" type, consisting of five digit numbers. We define a clipboard variable to be consistent if its value matches its specified data type.

Robofox incorporates three automated assertion generation mechanisms to cover these three classes of assertions. We describe these generation mechanisms next.

3.1.1 Action-based Assertion Generation

The most basic type of automated assertion generation mechanism provides pre and postconditions for each macro action following a predefined template. The preconditions ensure that conditions necessary for proper execution of the action are met, while the postconditions verify that actions performed by the macro produce their expected results. The considered actions and their corresponding pre and postconditions are listed in Table 2. The process for generating these assertions consists of traversing the WMG of a recorded macro and deriving the assertions for each action node by instantiating the corresponding pre and postconditions.

We illustrate this assertion generation process using three actions from Table 2. First, consider a *clip* action that extracts data from a webpage, assigns it a name, and places it on the clipboard. Before executing the *clip* action node, Robofox assumes that the element to be extracted is present on the webpage and that the clipboard does not contain an item with the same variable name. Thus, two precondition assertions are generated: the first is an existence assertion that verifies whether the extracted element can be found on the currently loaded webpage and the second is a non-existence assertion that verifies that the new element will not collide with an existing element in the clipboard. After the *clip* action is complete, Robofox assumes that the clipboard contains a new variable and its value is not empty. Thus, two postcondition assertions are generated: an existence assertion that verifies the availability of the clipboard variable and a value assertion that verifies that the variable's value is non-empty.

Next, consider a *filter* action that extracts a portion of some clipboard data (e.g., the day from a date field). Before executing the *filter* action, Robofox assumes that the clipboard variable used by the *filter* action is available. Thus, one precondition assertion is generated: an existence assertion that verifies the availability of the variable. After the *filter* action is complete, Robofox assumes that the value of the filtered data has the specified type. Thus, a postcondition assertion is generated: a type assertion that verifies that the type of the clipboard data is consistent with the specified type.

Finally, consider the *paste* action that places the content of a clipboard variable onto a webpage form field. Robofox assumes that the clipboard contains the variable and the target webpage form field is available before executing the *paste* action. This leads to the creation of three preconditions. After the *paste* action is performed, Robofox verifies the postcondition that the values of the clipboard variable and the webpage form field are identical.

Table 2: Actions and their assertion templates.

Action (parameter)	Precondition (class)	Postcondition (class)
Check (*elm*)	Element *elm* exists (existence)	Element *elm* is checked (value)
Choose (*elm*)	Element *elm* exists (existence)	Element *elm* is chosen (value)
Clip (*elm, var*)	Element *elm* exists (existence) Clipboard does not contain *var* (non-existence)	Clipboard contains defi ned variable *var* (existence) Value of clipboard *var* equals value of element *elm* (value)
Defi ne (*var, val*)	-	Clipboard contains the defi ned variable *var* (existence) with *val* (value)
Enter (*elm, val*)	Element *elm* exists (existence)	Value of element *elm* is *val* (value)
Filter (*var*)	Clipboard contains variable *var* (existence)	Clipboard contains subset of *var*'s value (existence)
Go Back ()	Browser has a previous history (existence)	Page load event is captured (existence)
Go Forward ()	Browser has a forward history (existence)	Page load event is captured (existence)
Import (*fi lename*)	File *fi lename* is accessible (existence) Clipboard does not contain *var(s)* (non-existence)	Clipboard contains *var(s)* (existence)
Load (*url*)	-	Page load event is captured (existence) Current URL is *url* (value)
Paste (*elm, var*)	Element *elm* exists (existence) Clipboard contains variable *var* (existence)	Value of element *elm* is equal to value of clipboard *var* (value)
Select (*elm, val*)	Element*elm* exists (existence)	Selected option of element *elm* is *val* (value)
Uncheck (*elm*)	Element *elm* exists (existence)	Element *elm* is not checked (value)
Wait (*sec*)	-	Wait time *sec* is elapsed or a page load event is captured (existence)

Algorithm 1 Generate Form Default Value Assertions

Input: *wmg*: graph representation of current web macro;
f: web form to be submitted.
Output: *assertions*: set of default value assertions.

```
1: assertions ← {}
2: action ← start-action
3: untouchedformFields ← f.getFieldsOfForm()
4: while (action ← wmg.getNext(action)) ≠ {} do
5:    if action.isModifyingAction() then
6:       modifiedField ← action.getField()
7:       untouchedformFields.remove(modifiedField)
8:    end if
9: end while
10: for all field in untouchedformFields do
11:    assertions.add(field.name, field.value)
12: end for
```

3.1.2 Forms' Default Value Assertion Generation

Forms' default value assertions ensure that the values of fields that were not modified by the user actions in the recording session are the same when the submit action is executed in a replay session. Default value assertions enable the prompt detection of failures caused by changes in a form's default values, such as the one described in the insurance quoting scenario in Section 2.1. This mechanism analyzes the fields of the submitted form, determining those that have not been modified by the macro's actions between the times at which the page is loaded and the form is submitted, and generates default value assertions for the unmodified fields.

Algorithm 1 provides details. The algorithm's first three lines initialize the assertion set and action, and obtain the list of all form fields that belong to the submitted form *f*. The algorithm then visits all action nodes in a WMG through the *getNext* method. If the visited action node modifies a field of the submitted form (e.g., enter, select, paste, check-field type actions), the algorithm removes the field from the set *untouchedformFields*. The traversal terminates when there are no more action nodes to process. When the loop terminates, the set *untouchedformFields* contains just the unmodified fields for which default value assertions are generated.

Robofox must execute this algorithm for all form-submission type actions. Since submission can be triggered by pressing a submit HTML object or by calling the JavaScript *submit* function, Robofox adds *onSubmit* event listeners and it overrides the native JavaScript's *submit* function to capture both types of submission. Similar event listeners and overriding functions are used by Robofox to trigger checking of the other assertions.

3.1.3 Clipboard Type Assertion Generation

A clipboard type assertion specifies that a clipboard element's value must match a certain pre-defined type. Enabling this class of assertion requires: 1) support to help users identify or specify at least certain basic types, and 2) an underlying mechanism for automatically propagating the type information to detect potentially conflicting or unsupported uses of the element based on its type.

Support for type specifications. Robofox provides a pre-defined library of basic and commonly understood type formats, such as Number, Date, and Currency, which most browser users can use to construct clipboard type assertions.

In addition to these basic formats, Robofox provides mechanisms by which more expert users can define custom types using *regular expressions* and *tope* formats. More expert users can employ regular expressions to do string matching as is done in popular scripting languages like perl. Tope formats complement regular expressions by allowing the validation of data through the combination of several facts about the data that individually may prove nothing about the data's validity, but together may suggest that an error has occurred. Consider the Data Migration scenario presented in Section 2.1 in which students' names were copied between websites. It is possible for last names to have 12 or more characters (e.g., "Grothendieck"), to contain spaces (e.g., "De Morgan"), or to start with lowercase letters (e.g., "von Neumann"). A regular expression for a last name would probably allow for each of these characteristics and would thus also accept "internal server error" as a name. But it is rare for a string to have all three of these characteristics. A tope format could note the multiplicity of unusual characteristics to detect the error.

To achieve this, a tope format allows users to specify soft constraints that are *often* satisfied by data. Users can associate constraints with the data string as a whole, or with parts of the string. Users can specify that data should contain, start with, or end with a number of specific characters (e.g., letters, digits, spaces, punctuation), or that data should be in a closed set of possible values. In addition, users can specify arithmetic constraints on values that should be treated as numbers. Each tope format is implemented internally as a context-free grammar with constraints attached to

productions. At runtime, the web macro tool takes the clipboard element's value as a string and parses it according to the grammar, then checks parts of the string against the constraints. Violations of constraints cause Robofox to downgrade confidence in the string's validity. Users can specify a threshold on how many such violations should be tolerated before the assertion fails.

Figure 3 displays the Tope Editor. Note how users can create a tope format without seeing the underlying context-free grammar; instead, our editor enables end-user programmers enter a list of examples from which the tool infers a boilerplate format that they can review, test, and customize. We have described and evaluated this algorithm in detail elsewhere [22]).

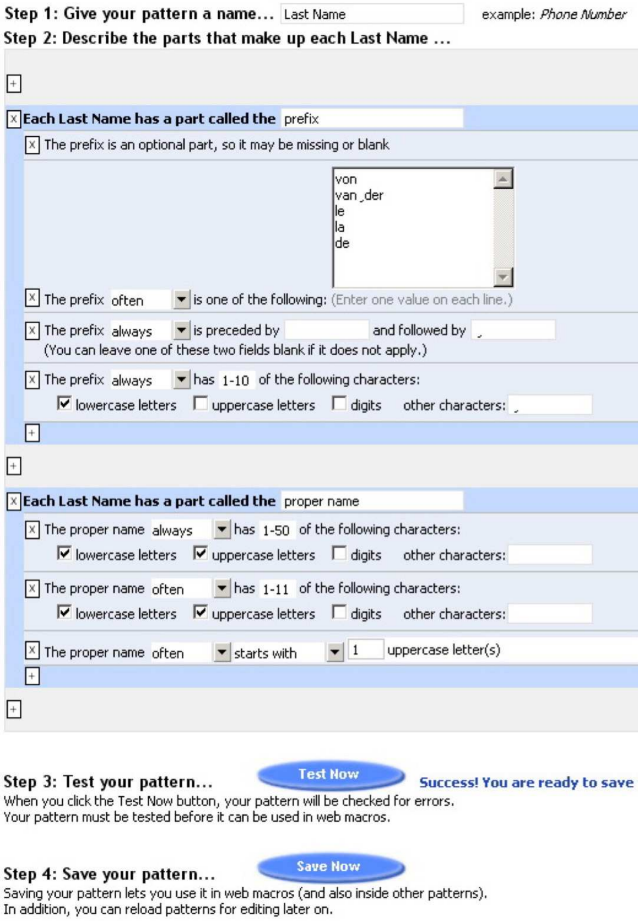

Figure 3: Tope Editor.

Type propagation. Once the user has specified a clipboard element type for a node in the WMG, Robofox automatically propagates that information to all nodes that access the element. This enables Robofox to detect potential inconsistencies in the use of the element at run-time. This is particularly useful in the presence of macros with branches, as there may be multiple definitions and uses throughout the macro that may introduce inconsistencies. The analysis we perform for type propagation is a forward, "may" dataflow analysis (using the union confluence operator), and is summarized by Algorithm 2. The input to the algorithm is the macro's WMG and the output is a set of assertions associated with each action node that uses a clipboard element.

The algorithm consists of two loops. In the first loop (lines 2-20) each action node is associated with all its reachable clipboard elements and their types. The algorithm traverses a macro's WMG from top to bottom (lines 4-19), adding to each node the clipboard

Algorithm 2 Generate Clipboard Type Assertions

Input: *wmg*: graph representation of current web macro.
Output: *assertions*: on clipboard type usages.

```
1:  noChangeInTypes ← 0
2:  while noChangeInTypes ≠ 2 do
3:    action ← start-action
4:    while (action ← wmg.getNext(action)) ≠ {} do
5:      temp-action.clipboardVars ← action.clipboardVars
6:      action.clipboardVars ← action.clipboardVars ∪
        wmg.getPrevious(action).clipboardVars
7:      if action.clipboardVars ≠ temp-action.clipboardVars
        then
8:        noChangeInTypes ← 0
9:      end if
10:     if action.isDefinitionAction() then
11:       action.addClipboard()
12:       noChangeInTypes ← 0
13:     else if action.isTypeDeclarationAction() then
14:       action.addClipboard(type)
15:       noChangeInTypes ← 0
16:     else
17:       noChangeInTypes ← noChangeInTypes + 1
18:     end if
19:   end while
20: end while
21:  action ← start-action
22: while (action ← wmg.getNext(action)) ≠ {} do
23:   if action.isUseAction() then
24:     typeCount ← action.countClipboardTypes()
25:     if typeCount = 1 then
26:       action.generateTypeAssertion()
27:     else if typeCount > 1 then
28:       Warning; potential clipboard type inconsistency
29:     end if
30:   end if
31: end while
```

variables and types from the previous node (union in line 6), identifying actions that set either a clipboard element (e.g., define action, clip action) or the type of a clipboard variable (e.g., filter action), and incorporating this information into the current action node. The first loop ends when no changes are observed on the collected clipboard type information across all action nodes, which is implemented through the *noChangeInTypes* construct. Note that, since a node may be accessed through multiple paths, multiple traversals are necessary to accumulate all the clipboard variables that may be read by each node. The second loop (lines 22-31) again traverses the WMG, generating assertions for all the nodes consisting of a "use" action (e.g., paste, export, filter). Each assertion will control at run-time whether the used element has the expected type. In addition, the algorithm produces a warning when more than one type is associated with a reachable and used clipboard element; this warning indicates the existence of at least one path with used clipboard variables of conflicting types.

3.2 Assertion Handling and Evaluation

Because the number of assertions generated by our techniques can be large (e.g., for the Data Migration scenario in Section 2.1 our techniques generate 113 assertions on 34 action nodes), by default, Robofox does not display assertions in web macros. End users using Robofox do not need to view or concern themselves with assertions unless those assertions are violated.

At times, however, more expert users may choose to view macros and assertions, and to facilitate this, Robofox attempts to improve the understandability of assertions and the macros that contain them by detecting repetitive patterns in macros and rolling those into loops. This abstraction helps macro understandability by reducing the number of individual assertions appearing in the macro file. It also allows a further simplification in which assertions within loops that are invariant with respect to loop iterations are removed from the loops and inserted into the macros prior to or after the loop. For example, in the presence of a loop containing a *clip* action within its body, Robofox replaces the individual collision assertions (described in Table 2) with an assertion at the end of the loop that checks whether the size of the clipboard after the execution of the loop is equal to the size of the clipboard before the loop plus the number of clips performed within the loop. This process also reduces the number of assertion evaluations that need to be performed during macro execution.

When executing a web macro containing assertions, Robofox performs an assertion evaluation for each visit to an action node with assertions. An assertion is violated when its boolean expression evaluates to false, indicating that an assumption that held in the recording session does not hold in the replay session. When an assertion is violated, Robofox alerts users that a violation has occurred by displaying an alert, as shown in Figure 4. The window describes the assertion violation and offers four options: (1) *Modify:* re-record the macro, (2) *Add Branch:* add an alternative execution, (3) *Show:* examine the action and assertion detail, and (4) *Cancel:* stop execution of the macro.

When a user presses the "Modify" button in Figure 4, Robofox starts a re-recording session. A re-recording session is similar to a recording session, but the replay progress up to the point of failure is preserved. Therefore, the user can record actions as in the recording session while maintaining the replay progress prior to the failure. When the user finishes re-recording, Robofox displays a dialog that shows the re-recorded macro. The user may choose to add the newly recorded actions to the macro or to replace a portion of the previously recorded macro with the newly recorded actions.

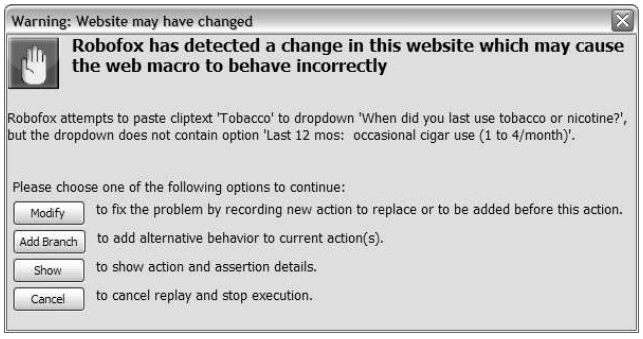

Figure 4: Robofox detecting an assertion violation.

The second option, "Add Branch," allows a user to incorporate an alternative execution branch to accommodate a new response from the target website that was not anticipated when the macro was first recorded. Robofox uses the violated assertion predicate as the starting condition for the newly created execution branch. The user can then perform a re-recording, similar to that of the previous option, but in this case, when the user completes the re-recording Robofox displays a dialog in which the user is asked to specify which nodes of the macro should be included in the alternative branch. A new execution branch creates a new alternative execution block, which starts with "alternative" and "alternative end" nodes. Consider again the Data Migration scenario (Section 2.1) in which a

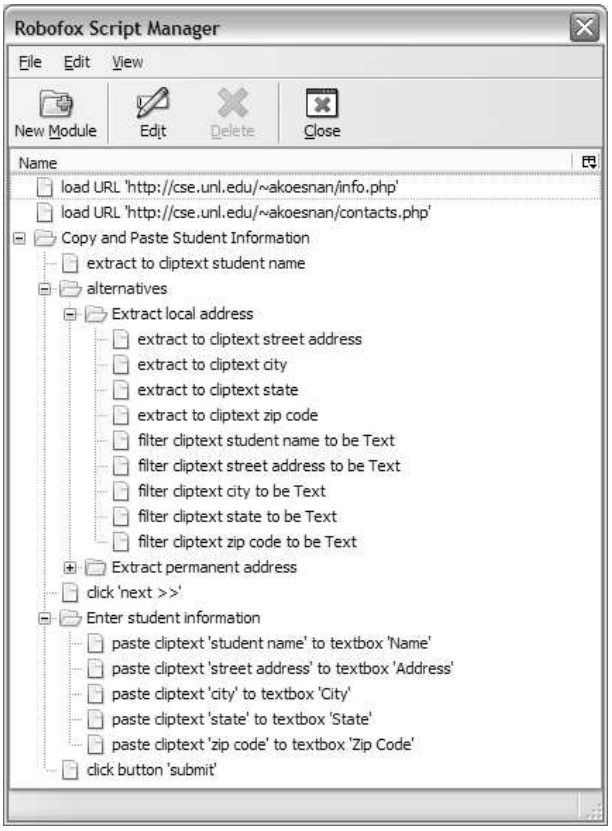

Figure 5: Setting an alternative execution branch.

student's address is copied between websites. Figure 5 illustrates how the script manager displays two alternative paths to extract the student address depending on whether the site provides a local or a permanent address (by default, the predicate that determines which branch to take is not displayed to the user to keep the script as simple as possible).

The last option for handling assertion violations, "Show," lets more expert users explore, edit, or delete violated assertions, allowing them to correct assertions that are invalid or too restrictive.

The final option, "Cancel," allows users to stop macro execution and edit the macro manually.

4. EMPIRICAL STUDY

To explore whether assertions in web macros can be useful, we performed an empirical study. The goal of the study was to analyze the *Robofox assertion mechanism* to *evaluate* it with respect to *effort and effectiveness to detect failures, identify their causes, and repair web macros* from the viewpoint of *researchers* in the context of *relatively skilled web macro users.*

The research questions we address in this study are:

- RQ1: Do assertions help users detect web macro failures?
- RQ2: Do assertions help users identify the cause of failures?
- RQ3: Does Robofox's re-record mechanism help users repair web macros?

4.1 Setup and Operation

As we have noted, there are a wide range of end users, from novice to expert, who may utilize web macros. For this study, we chose to consider relatively skilled macro users. We did this because (1) if relatively skilled users cannot make use of macro as-

sertions, then we have little hope that non-skilled users can do so, and (2) a population of such users was readily available from the ESQuaReD lab at the University of Nebraska – Lincoln, allowing us to more quickly conduct a formative study that could provide initial data on our approach, and inform the design of a future summative study on non-skilled users. We selected sixteen graduate students from the ESQuaReD lab as study subjects.

Ten of our subjects had been involved in a previous Robofox study before; this enhanced our ability to consider the subjects relatively skilled. Because our other six subjects had not had Robofox experience, however, we used a randomized block design to control for experience, first partitioning the students into blocks of high and low experience and then randomly dividing each block into control and treatment groups, with eight subjects per group. The treatment subjects were given a version of Robofox with the assertion and re-recording features enabled,[2] and the control subjects were given a version of Robofox with those features disabled.

As experimental tasks we chose two tasks corresponding to the two scenarios provided in Section 2.1): migrating student data from a university website to a bookstore website, and obtaining insurance quotes for a set of customers whose information is stored in a spreadsheet. These scenarios are actually based on observations of real web macro users [20]. They have the additional advantage of allowing us to make specific common modifications to test the use of assertions with, as detailed below.

The study was conducted over a period of two weeks. We initially provided the subjects with tutorial material on web automation, general concepts of PBD frameworks, and how to build web macros using Robofox, including exercises requiring its use. We used a questionnaire to determine whether the subjects had reviewed the tutorial and completed the exercises. Table 3 summarizes the subject's responses and shows that most had obtained exposure to Robofox either through the tutorial or from the earlier study.

Table 3: Previous exposure to Robofox.

	Reviewed tutorial or did previous study	Completed exercises or did previous study
Control	6 (75%)	5 (63%)
Treatment	7 (87%)	6 (75%)

The study then proceeded in two sessions, in which each subject performed web macro tasks with Robofox while being observed one-on-one by an experimenter (the first author). We expected each session to last no more than an hour but did not put any time constraints on the subjects. In the first session, subjects were asked to first build web macros for a warmup task not used in the remainder of the study, involving sending a stock quote to a cell phone via SMS. Following this, the subjects were asked to perform the two experimental tasks. In all cases the subjects interacted with mockup target sites that we developed based on real-world websites with similar functionality. Using these mockup sites let us to control for potential sources of noise due to variations across sites.

Following the first session, we applied a set of modifications to the mockup sites so that the macros would no longer perform correctly. These changes were selected based on historical changes made to similar websites as observed in the Internet Archive [7]. We chose simple rather than large, complex changes because detecting a large change in a web page is much easier than detecting more subtle changes, and thus our selected changes provided a more challenging test of our approach.

[2]Assertions generated through the three mechanisms discussed in the prior section were enabled. However, type specification was performed focusing only on Robofox built-in types.

Table 4 summarizes the four changes that we selected, one applied to the data migration task (F1) and three to the insurance task (F2, F3, F4). Each of these changes produced one failure, when considering these failures we call a failure an *apparent failure* if it causes the web macro execution to terminate prematurely during execution using Robofox with the assertion feature disabled. We call other failures *subtle failures*. As shown in the table, F2 and F3 are *apparent failures*, and F1 and F4 are *subtle failures*.

Table 4: Modifications to websites.

Failure	Class	Task	Web Site Change
F1	*Subtle*	Data migration	Displayed zipcode changes from 9 to 5 digit format
F2	*Apparent*	Insurance quote	Field type changes from radio button to dropdown
F3	*Apparent*	Insurance quote	Options available in dropdown change
F4	*Subtle*	Insurance quote	Default dropdown option changes

In the second session, the subjects were told to perform the same tasks they had performed in the first session, using the web macros they had built. They were told to imagine that a month had passed, and that the websites may have changed since when they first built the macros. They were also told to re-run their macros and to detect failures and fix the cause of the failures they encountered. Each subject was given a paper log form and asked to record: (1) the time at which they first encountered a failure, (2) the time at which they identified the cause of a failure, with a description of why the macro failed, and (3) the time at which they implemented a fix and verified its correctness with a successful execution. To reduce the risk of missing or incomplete data, the observer kept an independent record of this information. The observer also occasionally asked questions (e.g., "What are you thinking?") to determine what the subjects were doing when it was not obvious.

The subjects were permitted to ask the observer specific questions about features that they knew existed but had forgotten how to use, such as how to set macro execution speed or how to create a break point in a macro. However, questions about how to solve the tasks were not answered by the observer. At the end of the study, post-questionnaires were given to the subjects to capture their impressions of the study.

4.2 Results

We begin this section by depicting with boxplots (Figure 6) the time spent by the users attempting to detect each of the four failures, find the corresponding cause, and correct it. We observe that the control group required between one and 14 minutes longer than the treatment group to address each failure, and the difference was larger for "subtle" failures (the difference between control and treatment is 14 minutes for F1, and the control group subjects did not even detect F4).

In the following sections, we analyze these results in detail by separately examining failure detection, fault identification, and macro repair. For each research question, we present our null hypothesis or hypotheses and use statistical tests to accept or reject the null hypothesis with a 95% confidence level ($p<0.05$). We use the nonparametric Mann-Whitney test that does not make assumptions about the distribution of the data, and the Fisher Exact test to check for the equality of two frequency distributions for smaller samples.

A two-factor analysis of variance test on the time taken by subjects to build web macros in the first session indicated no statistically significant difference due to experience for time spent on the

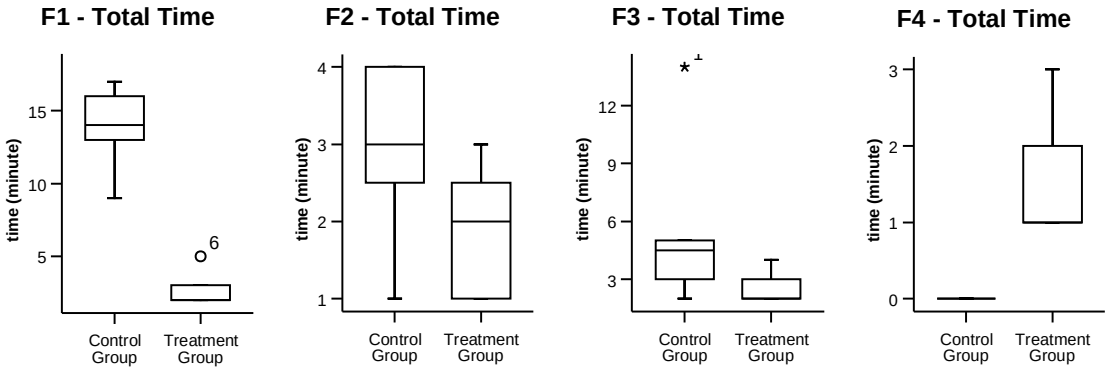

Figure 6: Time spent by the subjects to detect a failure, find the cause, and fix the macro (in minutes). Note that no subjects in the control group reported failure F4.

the insurance quote task ($p = 0.901$) or the data migration task ($p = 0.602$). Based on this, we decided not to consider experience as a factor in the remainder of our statistical analysis.

4.2.1 RQ1: Do assertions help users detect web macro failures?

To investigate the effect of assertions on the *users' ability* to detect failures, we compared the number of failures reported by the subjects. Our null hypothesis is: H_{0-1a}: *There is no difference between the number of failures detected by the two groups*.

Table 5 lists the subjects that detected each failure during the experiment. The eight subjects in the treatment group detected all four failures, whereas the subjects in the control group detected an average of 2.6 failures. The Mann-Whitney test indicates that treatment subjects detected statistically significantly more failures than control subjects with $p < 0.001$.

Table 5: Subjects detecting failures.

	F1	F2	F3	F4	Average
Control	5	8	8	0	5
Treatment	8	8	8	8	8

We also investigated the effect of assertions on the *users' effort* to detect failures. The null hypothesis is: H_{0-1b}: *There is no difference between the effort required to detect macro failures by the two groups*. To assess the effort required to detect a failure we measured the time elapsed from the beginning of the study to the time the user detected a failure. As mentioned in Section 4.1, these times were recorded by the subjects and verified by an observer, at minute-level resolution. The row of Table 6 labeled with RQ1 shows the average times taken to detect failures for each of the four faults considered, for the control and treatment groups. An *N/A* entry indicates that no subject was successful at performing the task.

There was no difference between the time spent to detect the simpler failures F2 and F3 in both groups, since Robofox showed a notification reporting it in less than a minute. However, on average, the control subjects required 2.6 minutes to detect failure F1, whereas the treatment subjects required less than one minute to detect this failure. While running the experiment we noted that the control subjects had to execute their macros multiple times to identify failure F1, with three subjects decreasing the web macro execution speed so that they could carefully examine whether each step was correctly performed. As shown in Table 7 under the "All Solutions" column, the Mann-Whitney test indicates that the control subjects spent statistically significantly less time to *detect* subtle failure F1 with $p = 0.001$. Therefore, we can conclude that assertions helped the users detect this more subtle failure and reduced the effort required to detect it.

Table 6: Average time spent to detect a failure, identify the cause, and repair the web macro (in minutes).

	Task	Group	F1	F2	F3	F4
RQ1	Detect Failure	Control	2.6	0.0	0.0	*N/A*
		Treatment	0.0	0.0	0.0	0.0
RQ2	Identify Cause	Control	7.8	1.6	3.5	*N/A*
		Treatment	1.6	1.2	1.6	1.1
RQ3	Repair Macro	Control	3.2	1.0	1.8	*N/A*
		Treatment	1.0	0.6	0.9	0.4

Table 7: Mann-Whitney tests of time spent for control vs. treatment. "*" denotes a statistically significant result.

		p-value	
		All Solutions	Correct Solutions
F1	Detect	0.001*	0.001*
	Identify	0.003*	0.003*
	Modify	0.004*	0.004*
F2	Detect	1.000	1.000
	Identify	0.268	0.903
	Modify	0.175	0.076
F3	Detect	1.000	1.000
	Identify	0.010*	0.032*
	Modify	0.179	0.046*

4.2.2 RQ2: Do assertions help users identify the cause of failures?

We next investigate the effect of assertions on the *users' abilities* to identify the causes of failures. The null hypothesis is: H_{0-2a}: *There is no difference between the effort required by the two groups to identify the causes for the macro failures.* To assess the effort required to identify the cause of a failure we measured the time elapsed from when the user detected a failure to the time they identified the reason for the failure (as recorded by the subject in a paper log and verified by the observer). The second non-header row of Table 6 (labeled RQ2) shows the average values for these times.

The control subjects required 4.9 times longer than the treatment subjects to identify the solution for failure F1, 1.3 times longer to identify failure F2, and 2.2 times longer to identify failure F3. None of the control subjects detected failure F4. The Mann-Whitney test indicates that the treatment subjects required statistically significantly less time to identify the cause for failures F1 and F3 with $p = 0.003$ and $p = 0.010$ respectively (Table 7 - "All Solutions" column). These results show that assertion violation messages helped users decrease the effort required to identify the solutions for web macro failures.

Table 8: Subjects reporting correct solutions.

	F1	F2	F3	F4	Average
Control	5	4	3	0	3
Treatment	8	7	6	7	7

We analyzed this data further by looking at the correctness of the solutions provided by the subjects and the effort required by the subjects to identify the correct solutions. The numbers in Table 8 show the number of correct solutions provided by the users for each failure. The null hypotheses are: H_{0-2b}: *There is no difference between the correctness of the macro modifications made by the two groups* and H_{0-2c}: *There is no difference between the effort required to identify the correct solutions for the macro failures by the two groups.*

The Mann-Whitney test across all failures indicates that the treatment subjects identified statistically significantly more correct solutions than the control subjects with $p = 0.005$. Similarly, the treatment subjects identified statistically significantly more correct solutions for the subtle failure F1 with $p < 0.001$. These results are consistent with the responses of the treatment group subjects on the post-test questionnaire, where seven out of eight subjects answered "Yes" to the question, "Did the assertion violation mechanism help you to identify the possible solution for failures?" giving reasons including the following:

- *"Sometimes the messages tell you what was expected. Knowing what was expected helps suggest a solution."*
- *"It shows differences and we can go and change the value. But sometimes we need to find the similar value instead."*
- *"It allows me to quickly identify the actual versus expected values or actions."*
- *"By saying what the value differences in the fields were, it helped me to identify the solution of the problem."*

However, there is no statistically significant difference for the *apparent failures*.

When considering only the correct solutions, the Mann-Whitney test (Table 7 - "Correct Solutions" column) indicates that the treatment subjects required significantly less time than the control subjects to identify the correct solutions for failures F1 ($p = 0.003$) and F3 ($p = 0.032$). These results are consistent with our previous findings and indicate that assertions helped our subjects identify the correct solution to macro failures faster.

The subjects' post-questionnaire responses show that there was a large difference between the subjects' opinions regarding the level of difficulty in identifying the correct solutions for the failures. Only one treatment subject mentioned that it was not easy to identify the correct solutions, whereas there were six control subjects who mentioned this. The Fisher Exact test shows that there is a statistically significant difference between their opinions with $p = 0.041$. Moreover, all subjects in the treatment group thought that the assertion violation messages were easy to understand and helped them identify the cause of the macro failures.

4.2.3 RQ3: Does Robofox's re-record mechanism help users repair web macros?

When Robofox detects a problem during a web macro execution, it offers users an opportunity to re-record the macro. For the treatment group, all four failures were detected automatically by Robofox. For the control group, only the two *apparent failures* gave them the opportunity to use the re-record option; for the *subtle failures*, the control subjects had to perform modifications manually using a drag-and-drop interface. This allowed us to analyze whether the re-record functionality reduced the effort required to perform modifications to the macro. The null hypothesis is: H_{0-3a}: *There is no difference between the effort required to modify the macro manually and using the re-record functionality.*

Of the five control subjects who reported failure F1, only one was able to perform modifications within a minute. The others spent between two and five minutes performing modifications. The Mann-Whitney test (rows labeled "Modify" in Table 7) indicates that the treatment subjects required statistically significantly less time to modify the macro than the control subjects with $p = 0.004$. Note that this significant difference was caused primarily by F1, but also by the additional information provided through the assertions for the other failures. In the post-questionnaire, all subjects indicated that the re-record functionality helped them modify their macros. Our experience working with the subjects leads us to believe that even if the subjects were more familiar with the tool, performing modifications without the re-recording feature would be difficult as this requires the subjects to set up break points, start recording sessions, and re-arrange the macros manually.

4.2.4 Findings Summary

The major findings of the study are as follows:

RQ1. Users employing the assertions mechanisms detected a significantly larger number of faults than users without assertions support (H_{0-1a}). Furthermore, users employing assertions detected subtle faults significantly faster (H_{0-1b}).

RQ2. Users employing assertions identified the cause of failures significantly faster (H_{0-2a}) and conducted the macro correction significantly more effectively and in less time than users not using assertions (H_{0-2b}, H_{0-2c}).

RQ3. Users with access to the macro re-recording mechanism were able to fix errors significantly faster than those implementing manual macro modifications (H_{0-3a}).

4.3 Threats to Validity

We now describe the threats to the validity of the study's findings and how we attempted to limit such threats.

External Validity. The web tasks and website changes we selected may not be representative of real tasks or changes. To limit these problems we created mockup websites that were similar to real world websites, and used changes similar to those observed on similar websites. Our subjects were Computer Science graduate students, most with professional software development experience, who know about testing and debugging programs. Thus, our results may not generalize to macro creators with lower levels of programming skill. However, our subjects never found it necessary to write any code during the study, and in this sense they did not have to rely on many of the programming skills at their disposal. Finally, in practice, users might run automated tasks in the background and may not be able to detect changes that do not cause the macro to terminate, whereas in our experiment the subjects observed the macro executions and were deliberately looking for failures.

Internal Validity. We conducted the study in a one-on-one setting so the subjects could not perform the tasks at the same time. Therefore, *history* effects cannot be ruled out. We attempted to minimize this by processing all of the subjects in a two-week period. Also, 6 out of 16 subjects had not participated in the earlier prior web macro experiment using Robofox and thus had less experience in using Robofox than the others. To address this we used a randomized block design. Our two-factor analysis of variance test of the time that the users spent in creating macros revealed no significant differences between experience groups

Construct Validity. Our earlier study revealed that subjects can be unreliable in recording their times so we had an observer record the timings as well. We ultimately used the observer's data for analysis. It was also this observer who evaluated the correctness of the identified failures. Since the observer was aware of which group the subject was assigned to, we cannot rule out bias in recording these values. It is also worth noting that we assumed the overhead associated with execution of the assertion generation and evaluation mechanisms did not perturb the subjects. Although we did not test that assumption within the experiment, preliminary runs on the scenarios described in Section 2.1 with and without the assertions mechanisms activated revealed overheads in the order of milliseconds per action. More specifically, for a macro like the one in Figure 5, the average execution time in replay using assertions was only 0.5 seconds slower than the average time without assertions. Finally, in this study we did not consider the effect of false positives on the usefulness of the assertion mechanism.

Conclusion Validity. Because of the small sample size (16 subjects), the absence of a statistically significant effect for factors (e.g., effort to identify causes and modifying macros for F2) may be due to insufficient power rather than the absence of an effect. Even with this small sample, however, it is clear that assertions enabled users to detect a significantly larger number of faults, to identify the causes of failure faster, and to fix errors faster.

5. CONCLUSION

Many tasks performed by web browser users are highly repetitive and fault-prone. Web macro tools such as Robofox can help these users automate repetitive web tasks but are still susceptible to errors caused by changes in and lack of contextual information about the web site on which the user operates. To help web macro users detect and fix such errors, we have developed assertion mechanisms and incorporated them into Robofox. These mechanisms hide the underlying complexity from the user who does not need to learn new technology, and they are tailored toward common types of errors encountered in web macros. Our study shows that Robofox's assertions can reduce the effort required for users to detect web macro failures, identify their causes and repair the macros.

We plan to extend Robofox's underlying mechanisms in several directions to further support web macro dependability. First, we are investigating the adaptation of existing inference mechanisms to derive types for items other than clipboard elements and to form repositories of types that can be shared among users of web macros. Second, we are studying opportunities for analyzing the WMG to accelerate macro execution by performing multiple web requests in parallel. This implies the need to determine what actions in the WMG nodes are independent and can be parallelized, but we expect that it can produce performance gains that enable the incorporation of more powerful macro analyses that we have not yet considered. Third, since web macros are often used in conjunction with other applications such as spreadsheets and word processors, we are identifying opportunities to gather and use the type and flow information available across these paradigms. Finally, we are investigating how to encode macros that abstract contextual information that may change when the macro is reused (e.g., abstract a particular date provided as input with information available in a calender, replace a fix list of macro actions with a loop that iterates a number of times in function of the size of the clipboard). We hope that such mechanisms will enhance macro robustness and extend their useful lifetime.

Robofox's documentation and source code are available for download at http://esquared.unl.edu/wikka.php?wakka=AboutRobofox.

Acknowledgments

This work was supported in part by NSF CAREER Award 0347518, the EUSES Consortium through NSF-ITR 0325273, NSF-CNS-0613823, and NSF-CCF-0438929. We would also like to thank the members of the ESQuaReD lab who participated in the study.

6. REFERENCES

[1] M. Bolin, M. Webber, P. Rha, T. Wilson, and R. C. Miller. Automation and customization of rendered web pages. In *ACM Symp. User Int. Softw. Tech.*, pages 163–172, 2005.

[2] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *Int'l. Conf. Softw. Eng.*, pages 93–103, 2003.

[3] CoScripter: Simplifying Web Processes. http://services.alphaworks.ibm.com/coscripter, Feb. 2008.

[4] A. Cypher. EAGER: Programming repetitive tasks by example. In *Conf. Human Fact. Comp. Sys.*, pages 33–39, 1991.

[5] Deskperience Web Replay. http://www.deskperience.com/webreplay, Jan. 2007.

[6] M. Erwig and M. Burnett. Adding apples and oranges. *Int. Symp. Pract. Aspects Decl. Langs.*, pages 173–191, 2002.

[7] Internet archive. http://www.archive.org, Mar. 2007.

[8] iOpus iMacros. http://www.iopus.com/imacros, Jan. 2007.

[9] iMacros Success Stories. http://www.iopus.com/imacros/success, Jan. 2007.

[10] Connect, collect, mashup everything on the web. http://www.kapowtech.com/products.html, Jan. 2007.

[11] T. A. Lau and D. S. Weld. Programming by demonstration: An inductive learning formulation. In *Int'l. Conf. Intelligent User Int.*, pages 145–152, 1999.

[12] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[13] Newbie Web Automation. http://www.newbielabs.com, Jan. 2007.

[14] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Int'l. Symp. Softw. Test. Anal.*, pages 232–242, July 22–24, 2002.

[15] HP QuickTest Professional. http://www.hp.com, Feb. 2008.

[16] IBM Rational Functional Tester. http://www.ibm.com/software/awdtools/tester/functional, Feb. 2008.

[17] A. Repenning and C. Perrone. Programming by example: Programming by analogous examples. *Comm. ACM*, 43(3):90–97, 2000.

[18] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995.

[19] G. Rothermel, M. Burnett, L. Li, C. Dupuis, and A. Sheretov. A methodology for testing spreadsheets. *ACM Trans. Softw. Eng. Meth.*, 10(1):110–147, 2001.

[20] C. Scaffidi, A. Cypher, S. Elbaum, A. Koesnandar, and B. Myers. The EUSES web macro scenario corpus: Version 1.0. Technical report, School of CS, Carnegie Mellon University, 2006.

[21] C. Scaffidi, A. Cypher, S. Elbaum, A. Koesnandar, and B. Myers. Scenario-based requirements for web macro tools. In *Vis. Lang. Human-Centric Comp.*, pages 197–204, 2007.

[22] C. Scaffidi, B. Myers, and M. Shaw. The Topes Format Editor and Parser. Technical Report CMU-ISRI-07-104, School of CS, Carnegie Mellon University, 2007.