RI14005 21 February 2014 Computer Science

IBM Research Report

Robust Test Automation Using Contextual Clues

Rahulkrishna Yandrapally
IBM Research – India

Suresh Thummalapenta
IBM Research – India

Saurabh Sinha
IBM Research – India

Satish Chandra
Samsung Electronics
USA

Abstract

Despite the seemingly obvious advantage of test automation, significant skepticism exists in the industry regarding its cost-benefit tradeoffs. Test scripts for web applications are *fragile*: even small changes in the page layout can break a number of tests, requiring the expense of re-automating them. Moreover, a test script created for one browser cannot be relied upon to run on a different web browser: it requires duplicate effort to create and maintain versions of tests for a variety of browsers. Because of these hidden costs, organizations often fall back to manual testing.

We present a fresh solution to the problem of test script fragility. Often, the root cause of test script fragility is that, to identify UI elements on a page, tools typically record some metadata that depends on the internal representation of the page in a browser. Our technique eliminates metadata almost entirely. Instead, it identifies UI elements relative to other prominent elements on the page. The core of our technique automatically identifies a series of *contextual clues* that unambiguously identify a UI element, without recording anything about the internal representation.

Empirical evidence shows that our technique is highly accurate in computing contextual clues, and outperforms existing techniques in its resilience to UI changes as well as browser changes.

1 Introduction

By test automation, we mean creation of executable *test scripts* that exercise a web application through a set of intended test scenarios. Here is a simple test scenario on the Skype website.

```
1: Click on Prices and Select Rates
2: Select "View rates" under "United Kingdom"
3: Verify that "Rates to United Kingdom" exists
```

This scenario can obviously be enacted by a human on a browser, but it can also be automated into a script. Table 1 shows the corresponding test scripts in several different test-automation technologies. Each script can be thought of as a simple program that contains description of an action to be performed on the application's GUI—typically a web browser—and some metadata that identifies the target of that action. We will explain these test-script notations and their limitations shortly.

1.1 The Fragility Problem

The promise of test automation is that once test scripts are created, they can be used for efficient regression testing cycles. Despite this promise, there are pragmatic reasons due to which organizations hesitate to deploy test automation and instead resort to ad-hoc manual testing, in which a human follows the steps of a test scenario and enacts them on the browser. The reason is that virtually all the existing automation technologies suffer from the *fragility* problem. Once created, a script breaks easily in response to minor changes, discussed next, causing both the expense of triaging script failures from real application failures, as well as the expense of updating the scripts—only to pay that expense again at the next slew of changes!

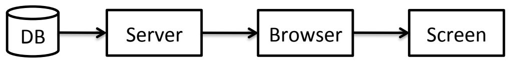
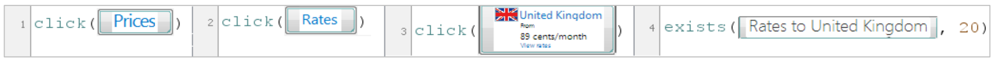


Figure 1: Flow of content to screen.

What are these changes? Figure 1 shows how what one sees on the screen is generated: the server reads some database state, composes a page (in HTML/CSS) and sends it to a browser. The browser translates the page into an internal representation, called the Document Object Model (DOM), and renders the DOM on a screen using a layout engine. Obviously, modern web sites are a lot more sophisticated; for example, the browser may ask the server for additional data before it completes the rendering, but this simple flow is sufficient to illustrate the changes.

First, the site designer could decide to change the look of the page, by changing the HTML/CSS that it sends to the browser. For example, the screen encountered at the third step of the Skype scenario could change from Figure 2(a) to Figure 2(b), in which four most-popular destinations are shown instead of three. Second, the database state may change: the three most-popular destinations—if generated automatically based on statistics—may get reordered. Third, the browser may represent the DOM in slightly different ways; this happens on a browser version upgrade and, of course, with a different browser altogether. Lastly, with browser changes, the layout for a given DOM may also change, but this is less common. In any event, most, although not all, test-automation technologies connect to the browser at the DOM level.

Table 1: Test scripts for the Skype scenario created using different automation techniques.

Automation Technique	**UI Element Identification Method**	**Test Script**
Record-Replay	Attribute, XPath, and CSS selectors	Command / Target / Value click / link=Prices / clickAndWait / link=Rates / click / //div[@id='popularDestinationRates']/div/div[3]/a/div[2]/span[2] / waitForText / css=h2.ratesTitle.title-h1 / Rates to United Kingdom
ATA	Label association, Detailed XPath	<Click, Prices, > xpath=/html/body/div/nav/ul/li[index:"6"]/a[href:"/en/rates/", title:"Prices"]/#text[text:"Prices"] <Click, Rates, > xpath=/html/body/div/nav/ul/li[index:"6"]/ul/li/a[href:"/en/rates/", title:"Rates"]/#text[text:"Rates"] <Click, View rates, > xpath=/html/body/div/.../div[class:"threeColumns separated"]/div[index:"2", class:"column"]/.../ #text[text:"View rates"] <Exists, Rates to United Kingdom, > xpath=/html/body/div/.../div[class:"ratesWidget section"]/.../ #text[text:"Rates to United Kingdom"]
ImageProcessing	Image	click(Prices) click(Rates) click(United Kingdom From 89 cents/month View rates) exists(Rates to United Kingdom , 20)
Descriptive Programming	Attribute selector	Browser("title:=Skype - Free internet...").Link("innertext:=Prices").Click Browser("title:=Skype - Free internet...").Link("innertext:=Rates").Click Browser("title:=Cheap international calls...").Link("innertext:=United KingdomFrom89 cents/monthView rates") .WebElement("innertext:= View rates", "class:=noMobile").Click If Browser("title:=Cheap international calls...").WebElement("innertext:=Rates to United Kingdom").Exist = false Then msgbox("Failed") End If
Our approach	**Contextual clues**	1. Click on "Prices" 2. Click on "Rates" 3. Click on "View Rates" near "United Kingdom" 4. Verify "Rates to United Kingdom" exists

The steps of a test scenario described in plain English are equally meaningful in the face of the changes described above. To a human, these changes present no trouble. Somewhat surprisingly, almost all the scripts shown in Table 1 will break because—as we elaborate in the next subsection—scripts bake in some form of the internal representation of the structure of a web page.

Of particular annoyance to organizations is the problem of script fragility across browsers, or even a different version of the same browser. This means that organizations must create and maintain scripts separately for all the browsers, and versions of those browsers, that their customers are likely to be using.

The challenge is, can a script work almost in the way a human perceives a web page, and is robust in the presence of the kinds of changes discussed above? The main contribution of the paper is a scripting notation that accomplishes this with high accuracy.

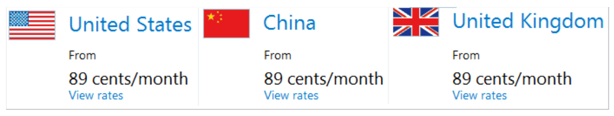
In the next subsections, we explain the script representations used by current automation technologies, and then show why even simple changes cannot be accommodated. We then outline our solution to this problem.

1.2 Existing Test-Automation Technologies

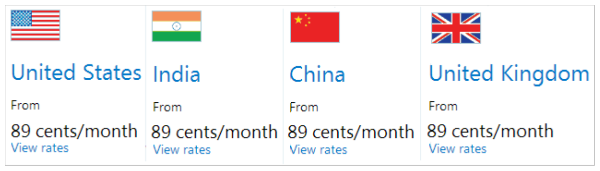
Record and Replay The simplest approach for test automation is *record-replay*, in which a tester performs the test steps via a testing tool (*e.g.,* [2, 3, 4]), the tool records the tester's actions, and creates a test script that replays the actions automatically. The first row in Table 1 shows the test script for the Skype test scenario recorded using Selenium [4]. The script specifies the actions to be performed and some information about the target UI elements.

In record and replay, the target UI element can be identified using different "selectors" that specify attributes, XPaths, or CSS paths. In the first two steps in the recorded script, the target is identified using the HTML `link` attribute. In the third step, the target is identified using an XPath selector, which is a path to the node for the target element in the DOM. Figure 3 shows a partial DOM for the Skype screenshot of Figure 2(a), rooted at the `div` element for `popularDestinationRates`. In the XPath selector, the part `//div[@id='popularDestinationRates']` locates the `div` element whose `id` attribute has the value `popularDestinationRates`; the remaining XPath encodes the path relative to this `div`. An XPath encodes both the hierarchy of elements and the relative position of an element within its parent; for example, `div[3]` refers to the third `div` element within the parent `div`. Finally, the fourth step records the target using a CSS selector.

The ATA Approach ATA is a research tool, developed in our previous work [22, 23], that attempts to address the perils of hard coding of paths in a DOM. ATA associates neighboring labels with UI elements and attempts to locate UI elements based on the associated labels. Thus, by not relying on DOM attributes and paths, it can relocate UI elements, even when they are moved or their internal attributes change, as long as the labels do not change. This solution works



(a) Original version



(b) Modified version

Figure 2: Example of a simulated change at the Skype website.

fine if the target label is unique within a webpage, but it is inadequate when a page contains multiple instances of a label (*e.g.,* the three occurrences of `View rates` in Figure 2(a)).

In such cases, ATA records the XPath of the target UI element alongwith some attributes and values for each XPath element. The second row in Table 1 shows the script generated by ATA. For the third step, which clicks on the `View rates` link, ATA records the XPath alongwith attribute information about the path elements: *e.g.,* for the third `div` element, ATA records the values of the `index` and `class` attributes. During playback, if the UI element cannot be located directly using indexes in the XPath, ATA first attempts to relocate the element via label association and also filters the identified elements based on the desired element type (*e.g.,* `#text`) described in the XPath. If it finds multiple elements of the same type, it recursively compares the attribute-value pairs of those elements and their parents, with the information stored in the XPath.

Image-Processing-Based Automation Another category of test-automation techniques relies on image processing; example tools include Eggplant [1] and SIKULI [9, 24]. These tools record a test script as a sequence of actions performed on images. The tester grabs a portion of the screen centered around the element of interest; when necessary, the tester must grab some of the surrounding screen to include disambiguating context.

The third row in Table 1 shows the test script recorded using SIKULI. Note that the image for the third step includes additional contextual information, such as the label `United Kingdom`, to locate the desired UI element correctly, because there are multiple "view rates" links on the page. During playback, these tools locate the target image and perform the desired actions at the *hotspot* position specified in the image. Usually, the center of the image is considered as the hotspot, but any other position can also be specified as the hotspot. By being totally DOM-independent, these tools are tolerant to variations in the DOM representations.

General Programming Test scripts can be written in a general-purpose programming language. For example, the Rational Functional Tester (RFT) [3] lets test scripts be written in Java, whereas in HP QuickTest Professional (QTP) [2], scripts can be written in VBScript; Selenium [4] supports many languages, such as Java, C#, PHP, and Perl. These tools provide API for accessing the DOM and locating UI elements programmatically via attributes, XPaths, etc. QTP also provides language support for a methodology called *descriptive programming* where a script describes a UI element by specifying attributes, and the tool attempts to locate the element using the specified attributes. The descriptive programming row in Table 1 shows the test script for the Skype scenario written in the QTP descriptive-programming style. For instance, the `innertext` HTML attribute is used to describe links and UI elements.

1.3 Illustration of Script Fragility

Consider a simulated change in the Skype application, so that at the third step, one sees the screenshot as shown in Figure 2(b) as opposed to Figure 2(a), where four most-popular destinations are shown instead of three. Let us see how the scripts generated by different automation techniques fare on the modified application.

The script generated using record-replay clicks the third `View rates` link on the page, following the recorded XPath `div/div[3]/.../span[2]`. In the path, `div[3]` refers to the third `div` element within its parent element—in the screen of Figure 2(a), it corresponds to the `div` for United Kingdom, whereas in the screen of Figure 2(b), it maps to the `div` for

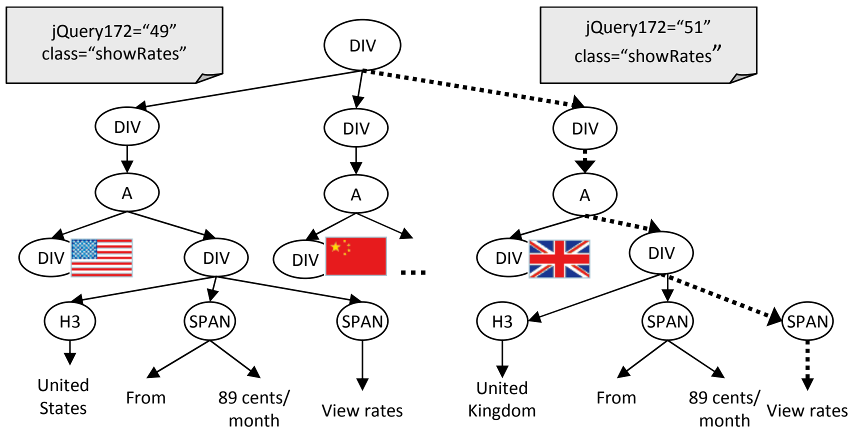


Figure 3: Partial DOM of the Skype webpage of Figure 2(a). The highlighted path leads to the instance of `View rates` referenced in step 3 of the sample test scenario.

China. `span[2]` refers to the second `span` element within its parent, and maps to `View rates` in either case. Thus, on the modified version, the script ends up clicking the `View rates` for China instead of United Kingdom.

ATA, in general, attempts to locate UI elements via labels to avoid fragile scripting that results from an over-reliance on XPaths, etc. However, for this example, ATA finds multiple UI elements when it attempts to locate the element at step 3 using the label `View rates`. Because of this ambiguity, it has to resort to the fallback method of locating the element via its XPath and the desired element type (`#text`). Thus, like record-replay, ATA erroneously narrows down to the `View rates` for China.

For the SIKULI script, image matching fails at step 3 because the destination reordering changes the location of the flag on the screen. (This change occurs automatically as the application rendering attempts to place four destinations in the space previously allocated for three destinations.)

Finally, the script created using descriptive programming can locate the correct `View rates` via the `innertext` attribute, whose value "`United KingdomFrom89 cents/monthView rates`" remains unchanged by destination reordering. However, if this value were to change, caused by changes in the Skype rates, the script would fail to locate the link. In fact, during the study, we noticed that the value of `innertext` attribute changed to `United KingdomFrom89 cents/monthFrom €1,02 per month View rates`, with no change in the actual visual appearance of the page. Because of this change, the script created using descriptive programming failed.

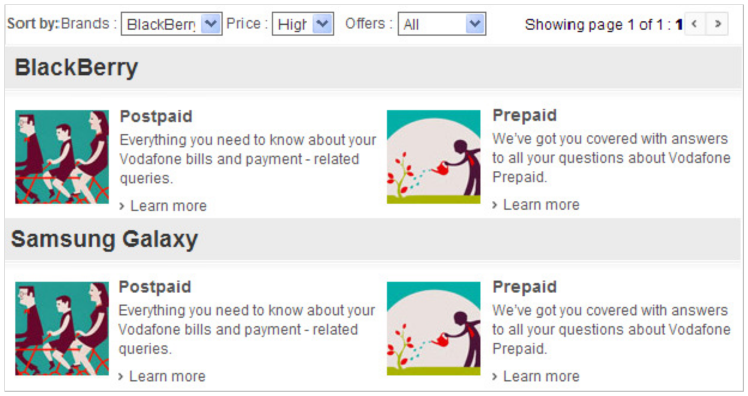
In general, it is possible to create a highly robust script using general programming, by writing error-handling code and attempting to locate a UI element using different strategies. But, creating such a script would entail a significant amount of coding. Moreover, such a script would still rely on DOM internals, which can differ across browsers. Anticipating such differences and writing browser-specific code would be even more tedious.

As an illustration of browser-specific differences in DOM, consider Figure 3. The two annotations on the `div` nodes show `jquery` attributes associated with those nodes, which appear in the DOMs rendered on Firefox and Internet Explorer (IE), but not in the DOM rendered on Chrome. Thus, a script that is automated on Firefox or IE and that refers to the `jquery` attribute would fail when executed on Chrome. To make scripts truly portable across browsers, such browser-specific differences must be known and the test scripts have to be carefully coded to avoid references to unreliable DOM attributes.

1.4 Our Approach

Although the change illustrated in Figure 2 breaks automated test scripts, it poses no problems to a human attempting to execute step 2 of the Skype test scenario: `Select "View rates" under "United Kingdom"` makes as much sense on the modified application version as it does on the original version. Thus, the main insight behind our technique is that an automated script can also rely on such contextual clues that a human uses. Such a script would be tool/platform-agnostic, highly portable, and tolerant to variations in DOM as well as differences in visual renderings.

The last row of Table 1 shows the test script generated by our technique. For step 3, the script contains the phrase "near United Kingdom," which identifies the intended `View rates` unambiguously ("near" is a keyword in our script notation). Our technique *automatically infers* such contextual clues or labels that are sufficient for locating the target


Sort by: Brands : BlackBerry Price : High Offers : All Showing page 1 of 1 : 1

BlackBerry

Postpaid
Everything you need to know about your Vodafone bills and payment - related queries.
› Learn more

Prepaid
We've got you covered with answers to all your questions about Vodafone Prepaid.
› Learn more

Samsung Galaxy

Postpaid
Everything you need to know about your Vodafone bills and payment - related queries.
› Learn more

Prepaid
We've got you covered with answers to all your questions about Vodafone Prepaid.
› Learn more

Figure 4: Screenshot illustrating ambiguous UI elements.

UI element uniquely. Moreover, as we show later, if a single clue is insufficient for locating an element, the technique can compute multiple clues that together identify the element of interest. To execute such a script, the technique *automatically* decodes the recorded contextual clues on the given DOM representation—which may differ from the representation on which the clues were constructed, *e.g.,* when the script is executed on a different browser than the one on which it was created—and identifies the element of interest. Note that no information about DOM structure is retained in our contextual clues or test scripts.

We implemented our technique in a tool called ATA-QV.[1] We conducted three empirical studies using 47 test cases (428 test steps) written for five public enterprise web applications. In the first study, we evaluated the accuracy of ATA-QV in computing contextual clues. Our results show that ATA-QV computed appropriate contextual clues for 98% of the test steps; moreover, 73% of the tool-computed clues matched exactly with the human-selected clues. In the second study, we investigated the change-resiliency of the scripts on real changes introduced due to evolution of applications and also on synthetic changes. Our results show that ATA-QV handled 95% of the changes (both real and synthetic) and outperformed existing state-of-the-art tools. In the final study, we investigated the portability of the scripts across browsers, by executing the scripts automated on Chrome on Firefox and IE. Our results show that 100% and 96% scripts passed on Firefox and IE, respectively.

1.5 Contributions

The contributions of our work are the following:

- A novel script notation in which a target UI element is identified using contextual clues—with "near" clauses—that are available on the page, without relying on the specificities of the underlying DOM or on image matching (both of which can lead to script fragility).
- A highly effective recording algorithm that, given a target UI element, automatically computes contextual clues for locating the element, even in the presence ambiguities. Our results show that, in about 75% of the cases, the algorithm computes the same contextual clues that a human would use.
- An efficient playback algorithm that replays a script created in this notation.
- An empirical evaluation of how various state-of-the-art test-automation technologies perform on real-world applications, in the presence of content changes and browser differences.

2 Our Technique

Like a conventional automation technique (*e.g.,* record-replay), our technique consists of two phases: the automation phase and the playback phase. The *automation phase* creates a test script of the form shown in the last row of Table 1, in which steps that deal with ambiguous UI elements have contextual clues recorded. We refer to the contextual clues

[1]QV stands for *qui vicino*, an Italian term that translates to "hereabout" or "near here," alluding to ATA-QV's capability of inferring contextual clues in the vicinity of a UI element that identify the element unambiguously.

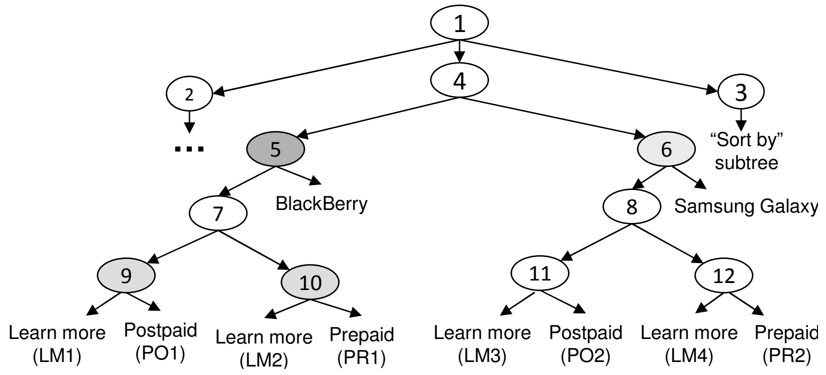


Figure 5: Partial DOM for the screenshot of Figure 4.

as *anchors*.[2] The *playback phase* executes a test script by performing each step of the test: it locates the target UI element by analyzing the recorded anchors and performs the required action on the element.

In this section, first, we present the core algorithms for automation and playback. Then, we explain the algorithmic extensions to handle cases where no suitable anchors exist (Section 2.3). To illustrate the algorithms, we use the example in Figure 4, which has four ambiguous instances of `Learn more` and each instance requires two anchors to be identified unambiguously. For example, the top-left instance is identified by anchors `Postpaid` and `BlackBerry`.

2.1 The Automation Phase

Given a target UI element e, the automation phase first creates a label to refer to e. For some types of UI elements, such as buttons or links, the text values inside those elements serve as the labels. For other types of UI elements, such as text boxes, combo boxes, or radio buttons, such text values do not exist. In such cases, the types of those elements (*i.e.,* "TextBox," "ComboBox," etc.) serve as the labels. If the initial label is unique, the technique does not need to identify any anchors—the initial label, by itself, is sufficient to locate e unambiguously. However, if the label is not unique, the technique has to compute the anchors for e.

The algorithm for computing the anchors (shown as Algorithm 1) works on the DOM representation of a webpage. Figure 5 shows a partial DOM for the screenshot of Figure 4. As shown, the root node has three children: the second child, node 4, includes the elements in containers of `Blackberry` and `Samsung Galaxy`; the third child, node 3, includes the subtree for the `Sort by` form. We omit nodes that are not relevant for explaining the algorithm.

The algorithm takes as input the *target label* l_i (*i.e.,* the initial label for the target UI element) and computes the anchors required for identifying l_i uniquely. Before discussing the steps of Algorithm 1, we explain intuitively the idea underlying the algorithm.

Suppose that the target is the `LM1` instance of `Learn more` in the DOM shown in Figure 5. There are three other instances of the label, denoted `LM2`, `LM3`, and `LM4`. Intuitively, the algorithm proceeds as follows. It identifies the largest subtree in which the target label is unique, and the largest subtrees that contain the other ambiguous instances but not the target. For the `Learn more` labels, these subtrees are t^9, t^{10}, and t^6. (We use the notation t^n to refer to the DOM subtree rooted at node n.) t^9 and t^{10} are the subtrees for `LM1` and `LM2`, respectively, whereas t^6 is the subtree for `LM3` and `LM4`. Next, the algorithm attempts to find a label in t^9 that distinguishes it from t^{10} and t^6. However, no such label exists. Therefore, it picks the ambiguous subtree that is closest to t^9—in this case, t^{10}—and identifies a label in t^9 that at least distinguishes t^9 from t^{10}. `Postpaid` is such a label in t^9, so the algorithm selects it. Now that the algorithm has found a way of distinguishing t^9 and t^{10}, it navigates upward to their common ancestor, node 7, and further expands the subtree to find the maximal subtree that does not include t^6, the remaining ambiguous subtree; this maximal subtree is t^5. At this point, the algorithm searches for a label in t^5 that distinguishes it from t^6, which it finds in the form of `BlackBerry`. Thus, `Postpaid` and `BlackBerry` are identified as the anchors for `LM1`.

Lines 1–5 First, the algorithm finds the other labels that match the target label l_i and cause ambiguity (line 1). Then, it computes the unique subtree for l_i (line 2). The subtree for each other ambiguous label is computed as the largest subtree containing that particular label and not containing the interesting label (lines 3–4). Function

[2]The term *anchor* is not to be confused with the HTML `<a>` tag.

Algorithm 1: The algorithm for computing anchors for the given label (associated with the target UI element).

```
Input: Label $l_i$
Output: Set $A$ of anchors for $l_i$
1  let $L_{other}$ be the other labels on the page that match $l$;
2  let $t_i$ = GetInterestingSubtree($l_i$);
3  initialize the set $T_{other}$ of trees;
4  foreach $l \in L_{other}$ do
5      add GetOtherSubtree($l$) to $T_{other}$;
6  while $T_{other} \neq \emptyset$ do
7      let $l_{distinct}$ = GetDistinctLabel($t_i, T_{other}$);
8      if $l_{distinct} \neq null$ then
9          add $l_{distinct}$ to $A$;
10         return $A$;
11     let $T_{cl}$ = GetClosestSubtrees($t_i, T_{other}$);
12     let $l_{distinct}$ = GetDistinctLabel($t_i, T_{cl}$);
13     if $l_{distinct} = null$ then
14         return $null$;
15     add $l_{distinct}$ to $A$;
16     $t_i$ = MergeSubtrees($t_i, T_{cl}$);
17     remove $T_{cl}$ from $T_{other}$;
18 return $null$;
```

`GetInterestingSubtree()`, not shown, takes as input a label l and identifies the largest subtree containing l such that l is unique in the subtree. The function traverses upward in the DOM, starting at l, until it reaches a node n such that the subtree rooted at n has more than one instance of l. At that point, it returns the previously traversed node. For example, for `LM1`, `GetInterestingSubtree()` returns t^9 because the subtree rooted at the parent of t^9 contains two instances of `Learn more`. Similarly, `GetOtherSubtree()` for `LM3`, returns t^6—the largest subtree that contains `LM3` but not `LM1` (the target label).

Lines 7–10 Next, the algorithm attempts to find a label in the subtree of interest, t_i, that distinguishes t_i from other subtrees. It does this by invoking function `GetDistinctLabel()`, which identifies all labels in t_i, and sorts them based on heuristics that give higher precedence to labels that are headers, have larger font sizes, or are in close proximity to l_i. This step may not find such a unique label. For example, when `GetDistinctLabel()` is invoked on t^9 and $\{t^{10}, t^{11}, t^{12}\}$, it finds no unique label in t^9 that distinguishes it from each subtree in $\{t^{10}, t^{11}, t^{12}\}$ because t^{11} contains all the labels that occur in t^9. In the next iteration of the loop in line 6, when `GetDistinctLabel()` is invoked on t^5 and $\{t^{11}, t^{12}\}$, it finds `BlackBerry` as the distinguishing label in t^5.

Lines 11–17 If the algorithm does not find a distinct label in t_i, it expands t_i by merging it with the closest other subtrees, say T_{cl}. The rationale behind expanding t_i is that this allows additional labels to be considered that can distinguish between the *expanded* subtree and the *remaining* subtrees (that contain ambiguous labels). However, while expanding t_i, we need labels that distinguish between t_i and subtrees in T_{cl}. Without computing such labels, merging t_i with subtrees in T_{cl} would not make sense because the ambiguous instances of l_i in t_i and T_{cl} would be indistinguishable during playback. Therefore, first the algorithm finds a label that distinguishes t_i from all subtrees in T_{cl}, and adds that label to the set of anchors (lines 12–15).

Next, the algorithm merges t_i with subtrees in T_{cl} by calling function `MergeSubtrees()` (line 16). `MergeSubtrees()` invokes `GetInterestingSubtree()` by ignoring other ambiguous labels in T_{cl}. After expanding the subtree of interest, the algorithm removes subtrees in T_{cl} from the set of ambiguous subtrees (line 17).

For our example, the algorithm computes `Postpaid` as the anchor that distinguishes t^9 from t^{10}, merges t^9 and t^{10} to form t^7, and expands t^7 to t^5. In the next iteration of the loop in line 6, it detects `BlackBerry` as the label that distinguishes t^5 from t^{11} and t^{12}. Using these anchors, it generates the command for clicking `LM1` as `Click on "Learn more" near Postpaid near BlackBerry`.

Algorithm 2: The algorithm for identifying the target label from the recorded anchors.

```
Input: Label $l_i$, set $A$ of anchors
Output: The target instance of $l_i$
1  let $L_i$ be the set of all instances of $l_i$ in the DOM;
2  SetInstances($l_i, L_i$);
3  foreach $l_a \in A$ do
4      let $L_a$ be the set of instances of $l_a$ in the DOM;
5      SetInstances($l_a, L_a$);
6  push $l_i$ and $A$ onto stack $S$;
7  while $true$ do
8      pop labels $l_1, l_2$ from $S$;
9      FilterInstances($l_1, l_2$);
10     if $S$ is empty then
11         $L_2 =$ GetInstances($l_2$);
12         if $|L_2| = 1$ then
13             return $L_2[1]$;
           return $null$;
14     push $l_2$ onto $S$;
15 return $null$;
```

2.2 The Playback Phase

The playback phase executes the actions in an automated test script. The main task for this phase is to locate the target label l_i, using the anchors recorded in the test script. The key idea of playback is to refine subtrees iteratively using anchors. In a sense, this phase does the reverse of what the automation phase does: whereas the automation phase expands subtrees, starting from the subtree that contains l_i, to collect anchors, the playback phase starts with the entire DOM tree and prunes out subtrees, using the anchors, to narrow down to the subtree that contains l_i. Intuitively, the playback phase attempts to find the smallest subtree that includes l_i and all anchors (say, $a_1, a_2, \ldots, a_k$). Once such a subtree is identified, it discards the last anchor a_k because that anchor has served its purpose of focusing the search to a specific subtree. Next, within that subtree, the playback phase finds the smallest subtree that contains l_i and the anchors $a_1, a_2, \ldots, a_{k-1}$. It continues in this manner, narrowing the search space and discarding an anchor in each iteration, until it finds a subtree in which l_i is unique.

For our running example, the technique first identifies t^5 as the smallest subtree that contains l_i (`Learn more`) and the two anchors (`Postpaid` and `BlackBerry`). Next, it discards `BlackBerry` and searches within t^5 for the smallest subtree containing `Learn more` and `Postpaid`. At this step, it narrows down to t^9, in which `Learn more` is unique. Thus, it locates `LM1` as the target label.

Although intuitively this approach makes sense, a naïve algorithm based on identifying subtrees can be exponential. Suppose that there are k anchors and that the DOM contains n instances of l_i and n instances of each anchor. Then, there are n^{k+1} possible combinations of l_i and the anchors. Moreover, the complexity of identifying the smallest subtree for a combination is $O(k * h)$, where h is the height of the tree. Thus, the complexity of the search in each iteration is $O(n^{k+1} * k * h)$ and there are k iterations.

We present a polynomial-time playback algorithm (shown in Algorithms 2 and 3) that processes anchors in pairs, instead of processing all anchors together. In the presentation of the algorithm, we assume the availability of a global store that maps a label to its instances and that can be accessed using functions `SetInstances()` and `GetInstances()`.

Lines 1–5 First, the algorithm identifies all instances of l_i and each anchor in the DOM, and stores them in the global store. For our running example, for which `Learn more` is the target label, the algorithm identifies four instances—`LM1`, `LM2`, `LM3`, and `LM4`—and stores them in the global store. Similarly, it identifies two instances of `Postpaid` (`PO1` and `PO2`) and one instance of `Blackberry`.

Lines 6–14 Next, the algorithm pushes the label of interest l_i and all anchors—in order—onto stack S. Figure 6(a) shows the initial state of the stack. The figure also shows the contents of the global store as the instances of each

Algorithm 3: The algorithm for filtering anchor instances.

```
Input: Label $l_1$, label $l_2$
1  initialize a two-dimensional matrix $M$;
2  $L_1 = \texttt{GetInstances}(l_1)$;
3  $L_2 = \texttt{GetInstances}(l_2)$;
4  foreach $i = 1, \ldots, len(L_1)$ do
5      foreach $j = 1, \ldots, len(L_2)$ do
6          $M_{i,j} = \texttt{GetCommonAncestor}(L_1[i], L_2[j])$;
7  initialize $L_{\mathit{filtered}}$;
8  foreach $i = 1, \ldots, len(L_1)$ do
9      if $all\ values\ in\ M_i\ are\ equal$ then
10         continue;
11     let $maxval$ be the maximum value in $M_i$;
12     foreach $j = 1, \ldots, len(L_2)$ do
13         if $M_{i,j} = maxval$ then
14             add $L_2[j]$ to $L_{\mathit{filtered}}$;
15 $\texttt{SetInstances}(l_2, L_{\mathit{filtered}})$;
```

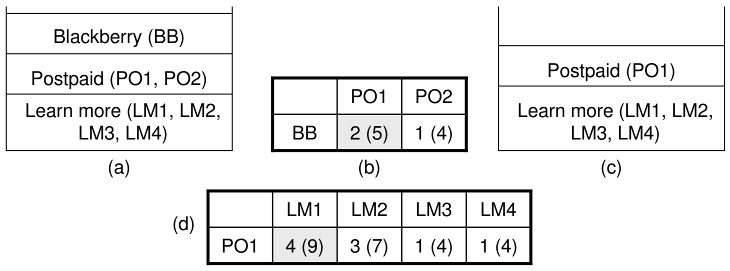
label on the stack. After initializing S, the algorithm iteratively removes the top-two elements, l_1 and l_2, from S and processes them. It invokes `FilterInstances` (explained later) to filter instances of l_2 using the instances of l_1 and store the filtered instances into the global store. When the stack becomes empty, the algorithm checks whether there exists only one instance of l_2 (the target label). If so, it returns the instance; otherwise, it returns `null` indicating failure to locate the target label.

FilterInstances (lines 1–6) `FilterInstances` contains the key steps for refining subtrees based on identifying the common ancestor of two given labels in the DOM. The function takes as input labels l_1 and l_2, and filters the instances of l_2 using the instances of l_1. For each pair of instances of l_1 and l_2, the function calls `GetCommonAncestor` to identify the farthest common ancestor from the DOM root node and compute the distance from the root. The function populates a two-dimensional matrix M with the computed distances. Figure 6(b) shows the matrix generated using the instances of `Blackberry` (`BB`) and `Postpaid` (`PO1` and `PO2`). Each cell in the matrix shows the distance of the common ancestor for that pair from the root; the values in parentheses are IDs of the common ancestors in Figure 5. For example, for `BB` and `PO1`, the common ancestor is node 5; it occurs at distance two from the root. Identifying the common ancestor for a pair of instances is analogous to identifying a subtree that is rooted at the common ancestor and that includes both the instances. For example, the common ancestor node 5 represents subtree t^5 that contains `BB` and `PO1`.

FilterInstances (lines 7–14) Next, `FilterInstances` analyzes each row in the matrix. If all values in a row are the same, it ignores the row (lines 9–10) because this indicates that the corresponding instance of l_1 do not help filter any instances of l_2. This could happen if there exist some instances of l_1 that are irrelevant for locating the target label. Next, the algorithm identifies the instances of l_2 with the maximum value in a row and adds them to the filtered list. The maximum value in the row indicates that the subtree of that common ancestor is smaller than the subtrees of other common ancestors. `FilterInstances` identifies all candidate instances of l_2 that belong to smaller subtrees and stores those instances in the global store.

For our running example, as shown in Figure 6(b), the algorithm selects `PO1` and stores the filtered instance in the global store. Figure 6(c) shows the state of the stack and the global store after the invocation of `FilterInstances`. Next, the main loop (loop 7–14) of Algorithm 2 invokes `FilterInstances` with l_1 set to `Postpaid` and l_2 set to `Learn more`, resulting in the computation of the matrix shown in Figure 6(d). Because the pair `PO1` and `LM1` has the highest distance value, the algorithm returns `LM1` as the target label.

Complexity Analysis Considering again the same parameters—n anchors with m instances of each anchor—algorithm `FilterInstances` takes $O(m^2 * h)$ time, where h is the height of the DOM. Because `FilterInstances` is called n times, the overall time complexity of the playback algorithm $O(n * m^2 * h)$.


Blackberry (BB)
Postpaid (PO1, PO2)
Learn more (LM1, LM2, LM3, LM4)
(a)
(b)
Postpaid (PO1)
Learn more (LM1, LM2, LM3, LM4)
(c)
(d)

	PO1	PO2
BB	2 (5)	1 (4)

	LM1	LM2	LM3	LM4
PO1	4 (9)	3 (7)	1 (4)	1 (4)

Figure 6: Illustration of label filtering during playback.

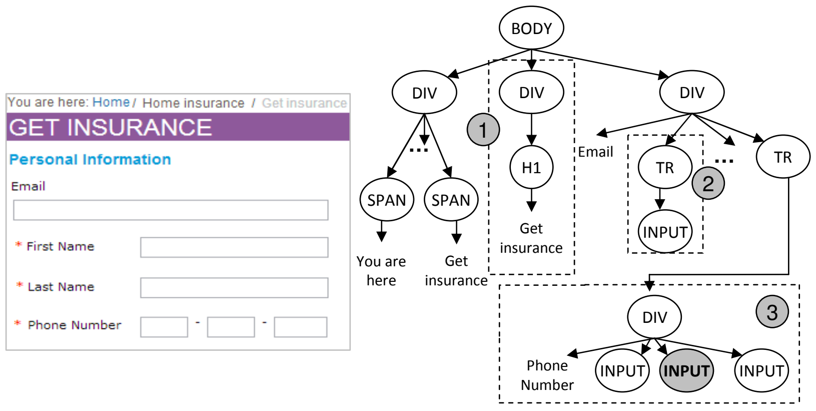


Figure 7: Illustrative example for the algorithm extensions: the screenshot of an insurance website and the associated DOM.

2.3 Extensions to the Algorithms

When the technique attempts to merge the subtree of interest (t_i) with the closest subtree (t_{cl}) and finds no suitable anchors to distinguish t_i and t_{cl}, it tries to compute three other types of anchors: negation anchors, proximity-based anchors, and ordinal-based anchors. We use Figure 7 to explain the extensions.

Negation Anchors Suppose that the `Get insurance` header in Figure 7 is the target element for a verification step. Because there exists another `Get insurance` label, in the bread crumb shown above the header, our technique attempts to find a label that distinguishes the subtrees containing the two instances. However, there is no other label in the subtree of the `Get insurance` header (shown as subtree 1). Therefore, the technique computes *negation anchors* to distinguish the two instances, and generates the script command `Verify that "Get insurance" not near "You are here" exists` to verify the existance of the `Get insurance` header.

Visual-proximity-based Anchors If the technique does not find suitable negation anchors, it tries to compute *visual-proximity-based anchors*: it searches for labels that are visually close to the target element in the left, bottom, right, and top directions. Suppose that the text box below `Email` is the target element. Because the corresponding subtree (shown as subtree 2) does not have any labels, the technique identifies nearby labels as anchors and generates the script as `Enter "t@gmail.com" in text box below "Email".`

Ordinal-based anchors If the technique does not find any proximity-based anchors, it next attempts to generate *ordinal-based anchors*. Whereas proximity-based anchors include only directional clues, ordinal-based anchors are more expressive and include both directional and positional clues. Consider the `Phone Number` label and the three text boxes next to it. Suppose that the target element is the second text box. The corresponding DOM is shown as subtree 3 in Figure 7. Neither negation nor proximity-based anchors can be computed for the target because the interesting subtree contains only the `INPUT` node (highlighted in subtree 3) for the second text box and there are no other nearby labels. Therefore, the technique computes an ordinal-based anchor by calculating the relative position of the text box

Table 2: Subjects used in the empirical studies.

Subject	Description	Test Cases	Test Steps			
			Tot	Act	Ver	Amb
Bell Canada	Phone, mobility, TV, and internet service provider (`www.bell.ca`)	9	119	87	32	57 (48%)
Bupa	Healthcare provider (`www.bupa.com.au`)	10	67	46	21	32 (48%)
Citi	Banking and financial services provider (`www.citibank.com`)	10	119	78	41	46 (39%)
ICICI Lombard	Insurance provider (`www.icicilombard.com`)	9	66	58	8	38 (58%)
Nationwide	Financial services provider (`www.nationwide.co.uk`)	9	57	34	23	25 (44%)
Total		47	428	303	125	198 (46%)

Tot:	Total steps	Act:	Action steps
Ver:	Verification steps	Amb:	Steps with ambiguity

from the nearest label in the left, bottom, right, or top directions, and generates the script `Enter "802" in the second text box to the right of "Phone Number"`.

3 Empirical Evaluation

We implemented our technique over the ATA tool [22, 23]; we refer to the enhanced tool as ATA-QV. Using ATA-QV, we conducted three empirical studies. In the first study, we evaluated the accuracy of ATA-QV in computing anchors. In the second and third studies, we investigated the resiliency of the scripts generated by ATA-QV to content changes and browser differences, respectively; in these studies, we also compared ATA-QV with existing tools.

3.1 Experimental Setup

We used five public enterprise websites as experimental subjects. Table 2 lists the subjects and, for each website, provides a brief description and the URL. The websites cover different domains, such as healthcare, insurance, and finance. All the sites host modern AJAX applications that exhibit a high level of dynamism and contain pages with ambiguous UI elements. Table 2 also presents information about the test cases for the subjects. We created the test cases to exercise different major functionalities offered by these websites. Column 7 shows the number and percentage of steps that reference ambiguous UI elements: the percentage of test steps that encounter ambiguity ranges from 39% to 58%.[3] For each test step that refers to an ambiguous element, we specified appropriate anchors in the manual test to identify the target element.

We used four tools: ATA-QV, ATA, SIKULI, and QTP. For each tool, we automated the tests on a *baseline browser*: Chrome 29 for ATA-QV, ATA, and SIKULI; Internet Explorer (IE) 8 for QTP (version 11 of QTP, which we used, does not support Chrome 29). On QTP, we used the record-replay feature to create the scripts; this feature of QTP has advanced capabilities for locating UI elements. For ATA-QV, ATA, and QTP, it is sufficient to point to the target element, and each tool automatically captures sufficient contextual information to identify the target element uniquely. However, for SIKULI, we have to ensure explicitly that the captured images include sufficient context to locate the target element uniquely, and also specify scrolling commands, where needed, to bring the relevant screen area into the view port. Therefore, automation using SIKULI took longer than the other tools. On average, for each subject, automation using ATA-QV, ATA, and QTP took about 30 minutes, whereas automation using SIKULI took 90 minutes.

Next, using each tool, we executed each script on the baseline browser. For ATA and SIKULI, all scripts executed successfully, whereas, for ATA-QV and QTP, one script for `Citi` failed. We used these results as the *baseline* for our studies and explain the reason for the failure in the subsequent sections. We also measured the total test execution time for each tool. On average, for each subject, QTP, SIKULI, and ATA took roughly the same amount of time ($\approx$14 minutes), whereas ATA-QV took more time ($\approx$20 minutes). This is expected as the playback algorithm of ATA-QV is more sophisticated and compute-intensive than that of the other tools.

[3]The test cases and test scripts are available at `https://sites.google.com/site/irlexternal/ata-qv`.

Table 3: Accuracy and types of anchors computed.

Subject	Anchor Comp	$A_t = A_h$	$A_t \neq A_h$ $\lvert A_t \rvert$ *Precise*	$\lvert A_t \rvert$ *Imprecise*	Types of Anchors Pos	Neg	Prox	Ord
Bell Canada	57	36 (63%)	21 (37%)	0	56	1	0	0
Bupa	32	20 (63%)	12 (37%)	0	27	4	1	0
Citi	46	35 (76%)	10 (22%)	1 (2%)	34	2	7	3
ICICI Lombard	38	35 (92%)	2 (5%)	1 (3%)	35	1	2	0
Nationwide	25	18 (72%)	7 (28%)	0	24	0	1	0
Total	198	144 (73%)	52 (26%)	2 (1%)	176	8	11	3

A_t: Tool-Computed Anchors A_h Human-Selected Anchors

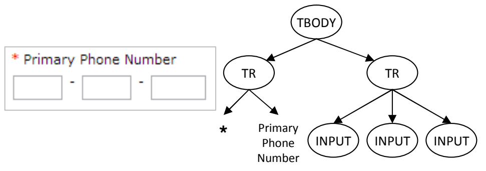


Figure 8: A screenshot from `Citi` showing three text boxes for entering phone number, and the corresponding DOM.

3.2 Study 1: Accuracy of the Technique

Goals and Method In the first study, we evaluated the accuracy of ATA-QV. For each test step that encountered ambiguity, we assessed the accuracy of the inferred anchors by comparing them with the anchors recorded in the manual test cases. An issue in this evaluation is that the ground truth (*i.e.,* the anchors recorded in the manual test) can be subjective: in cases where alternative anchors are available, different testers may select different anchors, or a tester may not specify all anchors. The anchors computed by ATA-QV are based on an objective analysis of the entire DOM (all of which may not be visible to the user at the same time) and guided by some heuristics (*e.g.,* ranking of candidate anchors based on font size, etc.); so, it is possible that tool-computed anchors differ from what a human might select. However, the tool-computed anchors guarantee mechanized identification of the target element unambiguously.

In the evaluation, we distinguished the cases where the tool-computed anchors (A_t) match, or differ from, the human-selected anchors (A_h). For the cases where the anchors differed, we further grouped them into two categories: *precise* and *imprecise*. The first category consists of cases where ATA-QV computed the minimum number of anchors required to identify the target element uniquely, but computed more anchors than the human. This happens if the human does not specify enough anchors in the manual tests. The second category consists of cases where ATA-QV computed more than the required number of anchors.

To investigate the applicability of the algorithm extensions (Section 2.3), we also identified the cases where no tree-based anchors could be computed and ATA-QV had to resort to computing negation, visual-proximity-based, or ordinal-based anchors.

Results and Analysis Table 3 presents the results of the study. Column 2 shows the number of steps that require anchor computation (as column 7 of Table 2 shows). Column 3 shows the number of steps for which A_t and A_h are the same, whereas columns 4 and 5 list the number of steps where A_t and A_h differ.

Overall, 73% (144 / 198) of the A_t anchors match exactly with the A_h anchors. Among others, interestingly, 26% (52 / 198) of the A_t anchors are precise. One reason is that, while writing manual tests, humans do not scan the entire web page to identify ambiguous instances, all of which may even not be visible, and then specify anchors. ATA-QV, however, scans the entire web page and generates the desired number of anchors to identify the target element precisely. Only in 1% (2 / 198) of the cases, ATA-QV computed more than the required number of anchors, which occurred due to the order in which the anchors are computed.

Columns 6–9 present data about the types of anchors computed. The results show that a large percentage of anchors—89% (176 / 198)—are tree-based anchors, showing the effectiveness of ATA-QV's core algorithm. A small percentage of anchors belong to the other three categories; ATA-QV computed these anchors due to lack of distinguishing labels within the subtrees of the target elements.

Over all subjects, ATA-QV successfully generated the script commands for 98% (419 / 428) of the test steps. For nine test steps that belong to one test for `Citi` (mentioned earlier as the failing test), ATA-QV could not generate relevant anchors. We explain the root cause for these cases using Figure 8, which shows a screenshot of a form containing three

Table 4: Types of real changes observed in the subject websites.

Change Category (Name)	Description	Exp. Result	# of Changes
Element type (EC)	Changes made to the type of a UI element	PASS	1
Attribute (AC)	Changes made to values of internal attributes such as `ID` or `name`	PASS	10
Structural (SC)	Changes where the position of the target element is modified (as illustrated in Fig 2)	PASS	19
Behavioral (BC)	Changes where the exercised scenario is changed, making the test obsolete	FAIL	10
Flow-level (FC)	Changes where the target label or test flow is changed	FAIL	8

text boxes for entering a phone number, and the corresponding DOM. For these text boxes, ATA-QV could not infer any tree-based anchor because all three text boxes, represented as INPUT elements, belong to one subtree, and the label `Primary Phone Number` belongs to a different subtree. Generally, in such cases, ATA-QV resorts to ordinal-based anchors. However, in this scenario, all the text boxes are at the same position with respect to the label—below the label. For such corner cases, ATA-QV falls back to the XPaths of ATA to generate the script command.

3.3 Study 2: Resilience to Content Changes

To assess resilience to content differences, we used two sets of changes: (1) *real changes* that were introduced during the evolution of the subject websites over a period of time, and (2) *synthetic changes* introduced in a controlled manner, by manipulating the browser DOM, for the purpose of this study.

3.3.1 Real Changes

Goals and Method To study real changes, we revisited our subject websites about four months after creating the initial scripts, and checked whether any changes were introduced in the pages exercised by the scripts. Table 4 summarizes the types of changes that we observed. The first three categories (EC, AC, and SC) represent internal DOM-level changes; *e.g.,* SC represents the changes where UI elements are moved within the page. In such cases, the scenarios exercised by the test cases are still valid and, therefore, the scripts are expected to pass (Column 3). An interesting aspect is that we found *no changes to anchor labels* identified by ATA-QV, which indicates that contextual clues may generally be more stable than DOM structures or internal attributes and, consequently, more robust for locating UI elements. The remaining two categories (BC and FC) represent semantic changes, where the covered behavior is modified (BC) or scripts require flow repair (insertion, deletion, or modification of steps) to execute successfully (FC). Therefore, scripts exercising such changes are expected to fail (Column 3) at the step, referred to as behavioral-difference-exposing step (*bde*), that encounters the behavioral difference. Column 4 of the table lists the number of changes in each category.

Using these changes, we classified the scripts into two groups: *passing* (group 1) and *failing* (group 2), which consist of 28 and 18 scripts, respectively. We ignored the script that could not be automated by ATA-QV (see Section 3.2). Next, we executed the scripts using each tool, on the baseline browser, and examined whether the expected test-execution results (pass or fail) were produced. For group 1, we consider an execution result as *true positive* if the tool successfully executes the script, or *false negative* otherwise. For group 2, we consider an execution result as: (1) *true positive* if the tool fails to execute the script exactly at the *bde* step, (2) *false positive* if the execution fails earlier than the *bde* step, or (3) *false negative* if the tool successfully executes the *bde* step. The false-negative result indicates that the tool failed to expose the behavioral difference. In summary, group 1 illustrates the change resilience of the tools in dealing with real changes, whereas group 2 illustrates the fault-detection capability (*i.e.,* the potential for exposing behavioral differences).

Results and Analysis Figure 9 presents the results: for each group and tool, it shows the percentage of scripts in each execution-result category. Overall, the results show that ATA-QV outperformed the other tools in both groups. Among the group 1 scripts, ATA-QV could not execute only two scripts. One of the scripts could not be executed due to change in the HTML element type—LABEL is changed to DIV (EC change category). Except SIKULI, none of the tools handled this change. The other failing script could not be executed because of a major structural change, which rendered the anchors useless in locating the target element. This change also modified the page layout, causing SIKULI and QTP to fail as well. The low true-positive rate of the other tools shows that they were unable to handle many of

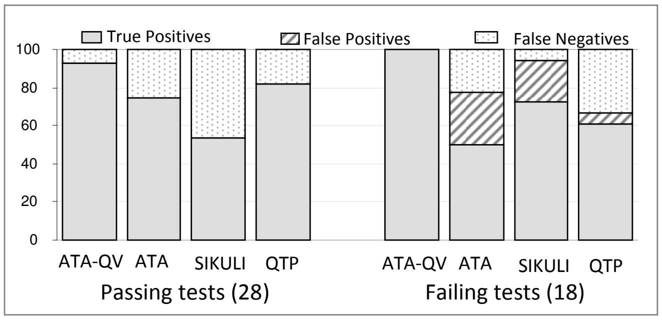


Figure 9: Results on resilience to real content differences.

the changes introduced in the course of normal website evolution; not surprisingly, test automation faces considerable skepticism in industry.

The group 2 results show that ATA-QV can be highly effective in exposing behavioral differences compared to the other techniques—ATA-QV exposed all the behavioral differences, whereas the other tools' effectiveness ranged from 50% (ATA) to 70% (SIKULI). Our results also show that QTP and ATA produced 33% and 22% false negatives, respectively, indicating that high reliance on DOM or internal attributes may end up performing actions on wrong UI elements at the bde step.

3.3.2 Synthetic Changes

Goals and Method To gain a deeper understanding of how each tool handles the AC and SC categories of changes (Table 4), we performed a controlled study using a set of synthetic changes. We used one subject, `Citi`, in which we introduced changes on specific pages by manipulating the browser DOM. For some test steps, we made both types of changes (represented using MIXED)—*i.e.,* we changed the position of the target element and also modified its attributes. We also introduced SC changes on pages where the target element is unambiguous. We use SC_UNB and SC_AMB to refer to changes in pages where target element is unambiguous and ambiguous, respectively. Note that all scripts are expected to pass for these synthetic changes. In all, we introduced 20 changes.

Results and Analysis Figure 10 presents the results: for each change category and tool, it shows the percentage of tests that executed successfully. The number next to a category name represents the number of changes in that category.

For the AC category, all tools except QTP successfully handled all the changes. Because QTP primarily relies on attributes for identifying target elements, it could handle only 20% of the changes. For SC_AMB, where the target element is ambiguous, only ATA-QV handled all the changes. Finally, for the MIXED category, only ATA-QV handled all changes except one change. SIKULI handled only 40% of the changes because changing some attributes, such as `align` or `size`, modifies the visual appearance of an element, resulting in SIKULI's image matching to fail.

Figure 11 illustrates a change that could not be handled by any tool. For the test step `Enter Zip Code as 27608`, ATA-QV generated a proximity-based anchor in the script—`Enter 27608 in textbox below "Find Citi Locations"`—as it did not find distinguishing labels within the subtree of the `Zip Code` text box. Because the modification placed the `Address` text box below the label, ATA-QV entered 27608 in the `Address` text box. Unlike DOM-based anchors, proximity-based anchors are susceptible to such changes in the visual appearance of a page. Overall, our results show that ATA-QV handled 95% (19 / 20) of the changes, substantiating the claim that ATA-QV generates more change-resilient scripts than the existing state-of-the-art tools.

3.4 Study 3: Resilience to Browser Differences

In the third study, we evaluated our technique's resilience to differences in an application's rendering on different browsers.

Goals and Method For all tools except QTP, we executed the automated scripts on Firefox 17 and IE 8. For QTP, we executed the scripts on Firefox 17. For each tool, we measured the number of passing scripts in each browser. Due to

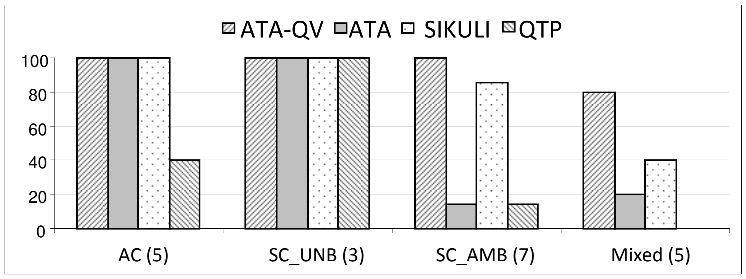


Figure 10: Results on resilience to synthetic content differences.

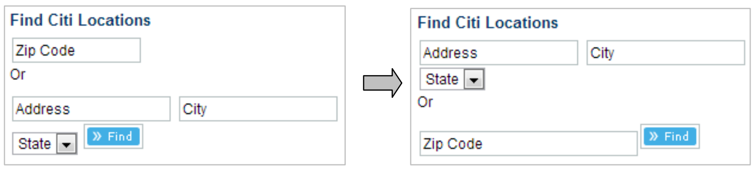


Figure 11: Example of structural and attribute changes.

limitations of Selenium in handling hover actions on IE [5], we could not execute ATA-QV and ATA for `ICICI Lombard`. Therefore, for ATA-QV and ATA on IE, we present the results for the remaining four subjects.

Results and Analysis Figure 12 presents the results. In the figure, the horizontal axis lists the subjects and tools, and the vertical axis shows the percentage of passing scripts. Next, we discuss the results for each tool and compare them with the baseline.

ATA-QV. Compared to the baseline results on Chrome, all scripts passed on Firefox . On IE, only two scripts failed in the four subjects (excluding `ICICI Lombard`). One of the scripts failed because of Selenium's limitation in handling frames in IE. For the other script, ATA-QV could not identify the target element because of a difference in the behavior of Chrome and IE. In particular, using Selenium on IE, some hidden popups—that are considered invisible in Chrome—are considered visible. One of these popups contains the target element and the computed anchors. Therefore, ATA-QV could not identify the target element.

ATA. On Firefox and IE, two and six scripts failed, respectively. The main reason is differences in internal IDs of parent elements, such as `DIV`, between Chrome, Firefox, and IE. Unlike ATA, most of the scripts passed with ATA-QV because ATA-QV does not rely on internal IDs.

SIKULI. In contrast to ATA and ATA-QV, a large percentage of the SIKULI scripts failed: 66% on Firefox and 62% on IE. There are two main reasons for the failures. First, different browsers render web pages differently. Although we attempted to adjust the similarity threshold to accommodate these differences, we noticed that reducing the threshold beyond a certain value resulted in spurious matches. For example, when we reduced the threshold to 0.5, an image captured for `First Name` matched with an image for `Last Name` because of the common "Name" term. Second, there are differences in scrolling effects across browsers for SIKULI. SIKULI (or any other image-matching-based technique) can identify elements only in the displayed view port of a web page. To handle parts that are not displayed, SIKULI provides commands, such as `WHEEL` or `PAGE_DOWN`, to scroll the page. However, different browsers respond to these commands differently, resulting in the script failures.

QTP. Finally, using QTP on IE as the baseline, five scripts failed on Firefox. The recent version of QTP incorporates a new feature that records the ordinal position of the target element along with its attributes. In our analysis, we found that many scripts passed due to this new feature. The script failures occurred primarily due to differences in internal IDs.

4 Discussion

Overall, our approach is a fundamentally new way of creating robust test scripts—by using contextual clues to locate UI elements. Its main strengths are as follows. First, it is resilient to changes in XPaths, values of internal attributes,

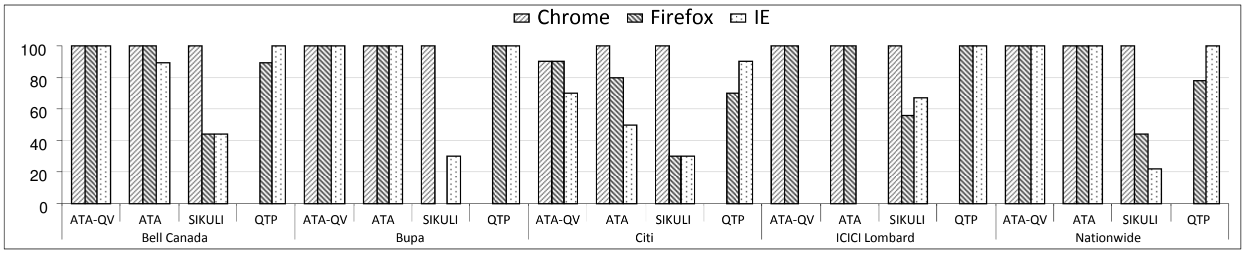


Figure 12: Results of the study on resilience to browser differences.

and differences in visual rendering. Second, it generates scripts that are platform-agnostic and tool-agnostic and, therefore, highly portable. Although image-matching-based techniques also result in somewhat agnostic scripts, in our experience (see Section 3.4), some platform-specific behavior, such as the amount of scrolling to be performed in different browsers, has to be coded in the scripts to make them truly portable.

In terms of weaknesses, the scripts generated by our techniques can be brittle in the presence of label changes, such as renaming a label or replacing a label with an image. Second, the DOM can be reorganized in ways that our technique is unable to play back a script, but where an image-matching-based technique would succeed. However, our empirical results show that these types of changes happen rarely compared to the other types of changes (which make the existing techniques generate fragile test scripts). Although further empirical evidence with more subjects will increase confidence in these observations, the fact that they are drawn from changes in production sites gives them considerable credence. Future studies, in more varied experimental setups, can investigate the generality of our observations.

5 Related Work

Corroborative Studies Several authors have written experience reports that mention the fragility of GUI tests. Berner et al. [7] present a compendium of practical advice on test automation based on their experience with real-world projects. They mention several challenges in test automation, among them the concern that GUI tests are hard to create and maintain—the latter because slight changes to UI can lead to adjustments in many scripts. Because of their tendency to break easily, GUI scripts often deliver a large number of false-positive failures and are, therefore, executed less often than expected, calling into question the investment in automation in the first place. Collins and Lucena [13] also found investment in automating GUI tests unproductive until the UI is stablized.

GUI Test Repair Because test maintenance is a frequent and time-consuming activity in the context of GUI test automation, several authors have proposed approaches to assist with this task.

Grechanik et al. [15] present a tool that identifies steps that are possibly impacted by changes in an application's GUI by comparing the GUI before and after the change. Their tool shows these steps to a tester for review and repair. WATER [12] is a technique for fixing broken GUI scripts. It records step-wise DOM states for the older, passing run of a script, as well as for a failing script run. Their technique tries to find, via exploration, alternative locations of UI elements or alternative content values that would make the script pass. These repair suggestions are shown to the user.

Memon et al. [18] present an approach for repairing GUI tests by automatically applying user-specified transformations to scripts. These transformations could, for example, insert a step that makes a previously broken script usable. Daniel et al. [14] propose to repair test scripts automatically by observing refactorings as they are being applied on a GUI and then deriving script refactorings from them. Recently, Zhang et al. [25] presented Flowfixer, an exploration-based technique to migrate scripts automatically to a new and evolved GUI. In comparison to these techniques, our approach creates scripts that are change resilient for a class of changes; the repair happens internally. However, the scope of change resilience is necessarily limited. Our technique will not, for example, automatically insert a new action step that was not necessary in the existing script but does become necessary with the changed GUI.

Languages and Notations Chickenfoot [8] is a programming system for automating web tasks that provides an API for finding elements on a web page by keyword matching on the label of the element. In case of ambiguities, positional clues such as "link after image" can be given. If there are multiple matches, Chickenfoot returns an iterator over all of the matches. CoScripter [16] is a system to create scripts based on observing user actions on a web page. It also includes positional attributes to help disambiguate an element of interest. ATA-QV's nesting of "near" clues is more expressive than these two systems. Moreover, ATA-QV relieves the user from having to identify the right contextual clues for precise matching.

Descriptive programming, as in QTP, also provides a "find" based API, with the difference that the API takes additional attributes against which to match an element. A programmer can use these API with a set of well-chosen attributes to locate the element of interest precisely. ATA-QV, by contrast, reduces this effort and can be used even by novices or non-programmers.

Image-Matching-Based Automation Similar to SIKULI [9], which we discussed in Section 1.2, Eggplant [1] is a commercial tool based on the principle of locating UI elements via image matching. As our experiments show, ATA-QV can be more robust than image-matching-based automation because image-based techniques can suffer from the differences in rendering and scrolling behavior.

Cross-Browser Testing WebDiff [10] analyzes web pages visually and detects layout differences across different browsers. Mesbah et al. [19] use dynamic crawling to analyze entire web applications and focus on detecting functional defects across browsers. CrossCheck [11] combines these two techniques to improve the detection of cross-browser incompatibility defects. These techniques are orthogonal to our technique and attempt to find functional and visual differences across browsers. These techniques can benefit from our technique by adopting our script representation.

Finally, other approaches that generate test cases for web applications (*e.g.,* [17, 20, 6, 21]) can also benefit from our technique in generating more change-resilient tests.

6 Summary and Future Work

We presented a novel solution to the script-fragility problem that widely afflicts the field of GUI test automation. Our solution generates test scripts by encoding contextual clues—instead of internal properties or visual renderings, as existing technologies do—for locating UI elements. The created scripts are tool/platform-agnostic and resilient to changes in internal properties or visual renderings. Our empirical results demonstrate the high accuracy of the technique in automatically inferring contextual clues. The results also show that the technique can be quite effective in compensating for real content changes and browser-specific DOM/rendering variations, and outperforms existing techniques. Thus, our technique can be a useful alternative to the existing approaches.

In general, there exist situations where any of these techniques can be better than the others and, in general, no one technique would be absolutely the best in all circumstances. We envision that, to address the script-fragility problem effectively, a tool would combine the different strategies for locating UI elements, providing users with flexibility to select the ones likely to generate the most robust scripts, taking into account various factors, such as the types of changes made in the application under test and the nature of testing performed (*e.g.,* functional regression testing, cross-browser testing, testing of locale-specific application variants, etc.). This would be a fruitful direction for future research.

Our approach may be even more valuable for automation of mobile applications. In that space, our platform-agnostic script notation, suitably extended, could be leveraged to deal with the high diversity (*e.g.,* in platforms, devices, and application variants) that poses challenges for robust and efficient test automation.

References

[1] EggPlant. `http://www.testplant.com/products/eggplant`.

[2] HP Quick Test Professional. `http://www8.hp.com/us/en/software-solutions/software.html?compURI=1172122&jumpid=reg_r1002_usen#.T-amcRfB-Gc`.

[3] IBM Rational Functional Tester. `http://www-01.ibm.com/software/awdtools/tester/functional`.

[4] Selenium. `http://seleniumhq.org`.

[5] Selenium Internet Explorer Driver. `https://code.google.com/p/selenium/wiki/InternetExplorerDriver/`.

[6] D. Amalfitano, A. R. Fasolino, and P. Tramontana. Rich internet application testing using execution trace data. In *ICST Workshops*, pages 274–283, 2010.

[7] S. Berner, R. Weber, and R. K. Keller. Observations and lessons learned from automated testing. In *Proceedings of the 27th international conference on Software engineering*, pages 571–579, 2005.

[8] M. Bolin, M. Webber, P. Rha, T. Wilson, and R. C. Miller. Automation and customization of rendered web pages. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 163–172, 2005.

[9] T.-H. Chang, T. Yeh, and R. C. Miller. GUI testing using computer vision. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 1535–1544, 2010.

[10] S. Choudhary, H. Versee, and A. Orso. WEBDIFF: Automated identification of cross-browser issues in web applications. In *IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[11] S. R. Choudhary, M. R. Prasad, and A. Orso. CrossCheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 171–180, 2012.

[12] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso. Water: Web application test repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, pages 24–29, 2011.

[13] E. F. Collins and V. F. de Lucena. Software test automation practices in agile development environment: An industry experience report. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 57–63, 2012.

[14] B. Daniel, Q. Luo, M. Mirzaaghaei, D. Dig, D. Marinov, and M. Pezzè. Automated GUI refactoring and test script repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, pages 38–41, 2011.

[15] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving gui-directed test scripts. In *Proceedings of the 31st International Conference on Software Engineering*, pages 408–418, 2009.

[16] G. Leshed, E. M. Haber, T. Matthews, and T. Lau. Coscripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1719–1728, 2008.

[17] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of AJAX web applications. In *International Conference on Software Testing, Verification, and Validation*, pages 121–130, 2008.

[18] A. M. Memon. Automatically repairing event sequence-based gui test suites for regression testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2):4:1–4:36, 2008.

[19] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 561–570, 2011.

[20] A. Mesbah and A. van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *ICSE*, pages 210–220, 2009.

[21] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra. Guided test generation for web applications. In *Proceedings of the International Conference on Software Engineering*, pages 162–171, 2013.

[22] S. Thummalapenta, N. Singhania, P. Devaki, S. Sinha, S. Chandra, A. K. Das, and S. Mangipudi. Efficiently scripting change-resilient tests. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering, Tool Demonstrations Track*, 2012.

[23] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra. Automating test automation. In *Proceedings of the 34th International Conference on Software Engineering*, pages 881–891, 2012.

[24] T. Yeh, T.-H. Chang, and R. C. Miller. Sikuli: Using GUI screenshots for search and automation. In *Proceedings of the 22nd Symposium on User Interface Software and Technology*, pages 183–192, 2009.

[25] S. Zhang, H. Lü, and M. D. Ernst. Automatically repairing broken workflows for evolving GUI applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 45–55, 2013.