Automated Modularization of GUI Test Cases

Rahulkrishna Yandrapally, Giriprasad Sridhara, Saurabh Sinha
IBM Research, India

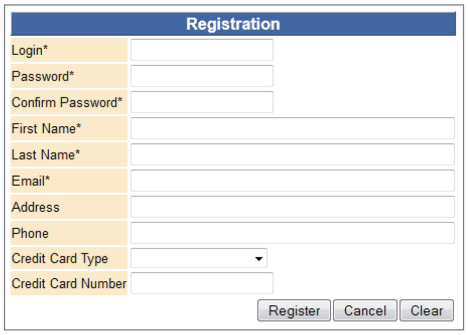
***Abstract*—Test cases that drive an application under test via its graphical user interface (GUI) consist of sequences of steps that perform actions on, or verify the state of, the application user interface. Such tests can be hard to maintain, especially if they are not properly modularized—that is, common steps occur in many test cases, which can make test maintenance cumbersome and expensive. Performing modularization manually can take up considerable human effort. To address this, we present an automated approach for modularizing GUI test cases. Our approach consists of multiple phases. In the first phase, it analyzes individual test cases to partition test steps into candidate subroutines, based on how user-interface elements are accessed in the steps. This phase can analyze the test cases only or also leverage execution traces of the tests, which involves a cost-accuracy tradeoff. In the second phase, the technique compares candidate subroutines across test cases, and refines them to compute the final set of subroutines. In the last phase, it creates callable subroutines, with parameterized data and control flow, and refactors the original tests to call the subroutines with context-specific data and control parameters. Our empirical results, collected using open-source applications, illustrate the effectiveness of the approach.**

I. Introduction

Test cases that drive an application under test via its graphical user interface (GUI) are frequently used in functional testing of enterprise applications. Such a test case consists of a sequence of steps that perform actions on the application user interface and check the expected behavior of the application, in response to those actions, as exhibited in the state of its user interface. GUI tests are created using a test-automation tool such as Selenium [1].

Like any software system, GUI test cases, after creation, require maintenance. As the application under test evolves, the test cases need to be adapted accordingly. Some tests could become obsolete and, therefore, are removed from the test suite, whereas other tests may need to be repaired—involving addition, deletion, or modification of test steps—to reflect the updated application behavior. In particular, content of the application screens might change, requiring the tests that navigate to those screens to be updated.

To ease test maintenance, it is essential that the tests cases are properly modularized. The benefits of modularization are well-known in software development, and these benefits are just as applicable to GUI test cases. To illustrate this, consider the Registration page of a `Bookstore` web application shown in Figure 1. There are several testing scenarios for this page. After providing the required data in the fields: (1) clicking the Register button results in successful registration, (2) clicking the Cancel button causes the registration process to terminate and the previous page to be displayed, and (3) clicking the Clear button erases all entered data and control stays on the



	t_1				t_2		
	Action	**Target**	**Data**		**Action**	**Target**	**Data**
1.	click	Registration		1.	click	Registration	
2.	enter	Login	L1	2.	enter	Login	L2
3.	enter	Password	P1	3.	enter	Password	P2
4.	enter	Confirm Password	P1	4.	enter	First Name	John
5.	enter	First Name	John	5.	enter	Last Name	Smith
6.	enter	Last Name	Smith	6.	click	Register	
7.	click	Register		7.	exists	⟨error message⟩	
8.	click	Sign in					
9.	enter	Login	L1				
10.	enter	Password	P1				
11.	click	Login					
12.	exists	⟨successful login⟩					

Fig. 1. The Registration page in the `Bookstore` web application (top). Two test cases that navigate to the Registration page (bottom).

Registration page. In addition to these (valid) scenarios, there are error scenarios to be exercised as well—attempting to register with an already existing login ID or without providing all required information—in which case, registration fails with an appropriate error message.

Figure 1 also shows two GUI test cases that navigate to the Registration page. Test t_1 exercises the valid scenario of successful registration, whereas t_2 covers the error scenario where registration is attempted without providing data for the required “Confirm Password” field. A test case is a sequence of steps where each step consists of an action, a target user-interface (UI) element on which the action is performed, and an optional data value. Step 12 in t_1 and step 7 in t_2 contain assertions that verify successful and failed registration, respectively (the actual messages are elided for brevity).

During application evolution, different structural and logical changes could be made that break GUI test cases [2]. As example of a structural change, the label “Login” could be changed to “User ID” which would affect whether the target UI element can be located in step 2 of t_1 and t_2. Partly, the problem of maintaining GUI tests pertains to the resilience of UI element locators to such structural page changes. A UI element can be located using different “selectors” such as XPaths, CSS paths, or HTML attributes (*e.g.,* `id` and `name`) [1] or, alternatively, via neighboring text labels [3] or contextual labels [4]. Depending on what locators are used and the types of changes made on a page, test steps may break and need to be patched. Apart from structural changes, logical changes that affect application flows can break test cases too: in such cases, steps have to be added or deleted to fix the test. For instance, the Registration page could be split into two pages where only the login text box appears on the first page, and control reaches to the second page (containing the remaining UI elements) only after the login value passes validation checks (*e.g.,* that the ID is not already existing).

The problem of test breakage caused by such structural and logical changes can be addressed partially by robust UI-element locators (*e.g.,* [4]) and test-repair techniques (*e.g.,* [5]–[8]), but these techniques nonetheless have limitations and, often, test repair requires manual intervention. In this situation, the manual maintenance effort would be much lower if tests were modularized. For our example, if the steps in t_1 and t_2 that are performed on the Registration page were extracted into a reusable module (which was invoked from those tests), changes on the Registration page would require updates in only the common module instead of each test case (and the Registration page, in fact, has more scenarios to be covered than just the two illustrated by t_1 and t_2).

These maintenance issues with GUI tests has led to development of the notion of *page objects* for creating modular and easy-to-maintain GUI tests [9], [10]. The idea is to encapsulate in a page object the UI elements and functionality on an application page, and then create test cases by invoking methods on page objects. The problem is that creating page objects is a manual process, requiring considerable human effort. Test scenarios have to be broken down into modularized sequences of steps based on the structure of the navigated pages, page objects with appropriate parameterized methods have to be created, and tests have to be coded by invoking the page-object methods. Leotta and colleagues [2] report that the development of GUI test cases via manual coding of modular page objects can be much more expensive than—in some cases, taking more than twice as much effort as—creating the same tests using capture-replay techniques (the latter set of tests, as expected, have a greater maintenance cost).

Therefore, an approach for automatically modularizing tests created via recording would be very useful: it would provide the benefits of low maintenance costs and avoid the upfront costs of crafting modular tests by hand. In this paper, we present such a technique. Our technique can be applied to GUI tests created via manual recording or automated test generation (*e.g.,* [11]–[15]), which also produce non-modular test cases.

Intuitively, modularization of GUI tests requires the structure of the navigated pages to be analyzed, so that elements on a "page" or, at a more granular level, in a "form" can be aggregated into a module. Thus, at a high level, our technique attempts to recover this structure by analyzing the UI elements accessed in the test steps. The notions of "page" and "form" are well-defined in conventional web applications: pages are associated with URLs and forms can be related to HTML `form` elements. But, in modern AJAX-style applications, which contain significant client-side processing, there is often no clear distinction between pages (*e.g.,* all application pages can have the same URL). Similarly, forms need not be associated with HTML `form` elements. To overcome these issues and to be more generally applicable, our approach analyzes the underlying Document Object Model (DOM) of the pages navigated by test cases. Our approach operates in three phases.

In the first phase, it analyzes each test case independently to partition the test steps into candidate subroutines. This phase analyzes the occurrences of the accessed UI elements in the DOM. It uses distance metrics on the DOM to detect the loading of a new page and, consequently, the beginning of a new module. This phase can be performed dynamically, by leveraging execution traces consisting of the DOM snapshot after each test step, or statically, by analyzing the test cases only. If DOM snapshots are available, the candidate subroutines can be computed more accurately (at the additional cost of trace collection). The output of this phase is a set of candidate subroutines, which feeds into the subsequent phases.

In the second phase, the technique compares candidate subroutines across test cases to compute the final set of subroutines. It creates initial groups of candidate subroutines and then refines the groups based on the degree of commonality in the accessed UI elements and whether the candidate subroutines in a group contain conflicting orderings of actions.

The last phase performs test refactoring to transform the original test suite into a suite consisting of reusable subroutines and modular test cases. Specifically, Phase 3 creates callable subroutines, with parameterized data and control flow, and replaces steps in the original test cases with calls to the subroutines along with context-specific data and control parameters.

Although many approaches have been developed for generating GUI tests (*e.g.,* [12]–[21]), there is not much existing research on modularizing GUI tests to make them easier to maintain, which is a problem with significant practical implications. Mahmud and Lau [22] present a supervised machine-learning technique for identifying subroutines in GUI test cases written in the ClearScript language [23]. Unlike that technique, our approach analyzes the structure of the pages navigated by a test case and does not require a training sample of subroutines. Section IV further discusses related work.

We implemented the technique and conducted empirical studies using four open-source web applications, with 179 GUI tests in total, to evaluate various aspects of the technique. The results show that our technique can extract highly reusable subroutines and reduce the sizes of test cases: for our subjects, it extracted 50 subroutines with an average of about six steps and eight call sites per subroutine; the total test steps decreased from over 1869 in the original test suites to 721 in the refactored suites (a reduction of 61%). Moreover, the extracted subroutines correlate closely to application pages, which typically is the goal when modular test cases are designed by hand, for example, using the page-object abstraction. This indicates that our approach could be effectively used to automate the task of finding plausible subroutines in recorded or automatically generated GUI tests.

The contributions of this work are

- The description of an automated technique for extracting subroutines from, and refactoring, GUI tests that is based on the structure of the pages navigated by the tests and is generally applicable.
- The implementation of the technique for modularizing GUI tests for web applications.
- The presentation of empirical results demonstrating the effectiveness and efficiency of the technique.

The rest of the paper is organized as follows. The next section describes our technique for modularizing GUI test cases. Section III presents the results of the empirical studies. Section IV discusses related work. Finally, Section V summarizes the paper and lists directions for future research.

II. Our Technique

Given a suite of GUI test cases, our technique refactors the tests by extracting reusable subroutines and replacing steps in the original tests with calls to the extracted subroutines.

Figure 2 shows the refactored `Bookstore` test cases and the extracted subroutine for entering registration information. To simplify the presentation, we use pseudo-code notation in Figure 2 to illustrate the subroutine and the tests; the implemented tool creates the refactored test suite as Selenium [1] test cases, written in Java, that can be executed using JUnit (see Section III). The extracted subroutine, `EnterRegDetails`, contains test steps that are executed on the Registration page and takes two input parameters: an array `data` of context-specific data values for test steps, and an array `control` of boolean flags controlling the execution of test steps.[1] The values of `data` are used in those test steps that required data: steps 1, 2, 4, 5, and 6. The boolean flags in `control` determine whether steps 4 and 9 are executed; the remaining steps execute unconditionally. Tests t_1 and t_2 are refactored to call `EnterRegDetails` (line 4 in t_1 and t_2), and set up the context-specific data and control parameters (lines 1 and 2 in t_1 and t_2). The data setup in t_1 creates five data values, whereas the data setup in t_2 creates four values—the third parameter, which is used for the "Confirm Password" text box, is empty because the corresponding test step (step 4 in `EnterRegDetails`) does not execute when `EnterRegDetails` is called from t_2. The control setup (line 2) ensures that when `EnterRegDetails` is invoked from t_1, step 4 executes and step 9 does not execute; the converse is true when `EnterRegDetails` is called from t_2.

Our technique performs this transformation in a mechanized way, and ensures that the behavior of the original tests is preserved in the transformed tests. Before presenting the algorithm, we provide some preliminary definitions.

[1]Although Figure 2 illustrates the extracted subroutine's data and control parameters as arrays, alternatively, a separate formal parameter with a meaningful name (*e.g.,* derived from attributes of the target UI element to which the parameter applies) could be generated for each data and/or control parameter.

```
   Sub EnterRegDetails(Str[] data, Bool[] control)
1.    enter, Login, data[1]
2.    enter, Password, data[2]
3.    if (control[1])
4.       enter, Confirm Password, data[3]
5.    enter, First Name, data[4]
6.    enter, Last Name, data[5]
7.    click, Register
8.    if (control[2])
9.       exists, error message
   End sub
   Test t1
1.    Str[] data = {"L1", "P1", "P1", "John", "Smith"}
2.    Bool[] control = {true, false}
3.    click, Registration
4.    call EnterRegDetails(data, control)
5.    click, Sign in
6.    enter, Login, L1
7.    enter, Password, P1
8.    click, Login
9.    exists, successful login
   End test
   Test t2
1.    Str[] data = {"L2", "P2", "", "John", "Smith"}
2.    Bool[] control = {false, true}
3.    click, Registration
4.    call EnterRegDetails(data, control)
   End test
```

Fig. 2. The transformed `Bookstore` test suite with the extracted subroutine for entering registration information, and the refactored test cases that call the extracted subroutine.

A. *Definitions*

We consider GUI test cases of the form shown in Figure 1(b). A *test case* is a sequence of test steps $\langle s_1, \ldots, s_n \rangle$, where each s_i is an action performed on the application user interface (an *action step*) or a verification of the user-interface state (a *verification step*). A *test step* is a triple (a, e, d) consisting of a command a, a target UI element e, and an optional data value d. The command a can be a predefined verification command (*e.g.,* `exists`, `selected`, `enabled`) that is a predicate on the state of a UI element, a predefined action command (*e.g.,* `right-click`), or a generic action command (*e.g.,* `enter`, `select`). The set of predefined commands depends on the test-automation tool being used.

The target UI element for a test step is identified using element locators. Web testing tools, such as Selenium [1], provide different types of locators that are based on XPaths, CSS paths, or HTML attributes. Our approach requires, in particular, the XPaths of UI elements. For example, for step 1 of tests t_1 and t_2, where the target is the "Registration" image on the `Bookstore` home page, the XPath `/html/body/table/tbody/tr/td/table/tbody/tr/td[3]/a/image` is recorded in those tests. An XPath encodes both the hierarchy of elements and the relative position of an element within its parent; for example, `td[3]` refers to the third `td` element within the parent `tr` element. Figure 3 shows a partial DOM for the Registration page in `Bookstore`. The UI elements, labels, and other HTML elements are nodes in the DOM.

To match test steps effectively across test cases, we use a notion of *equivalence*, rather than strict equality, which abstracts out unnecessary details. We define a *canonical representation* of a test step s as $\sigma(s) = [\![\sigma(a), \sigma(e), \sigma(d)]\!]$. $\sigma(a)$ maps command a to the constant `A` if a is a generic action command; otherwise, $\sigma(a) = a$. $\sigma(e)$ maps element e to its XPath. $\sigma(d)$ maps d to the boolean value `true` if d is non-

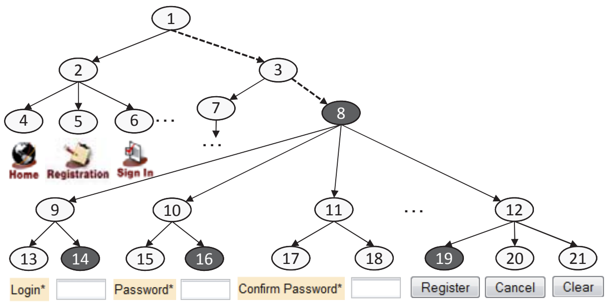


Fig. 3. Partial DOM for the `Bookstore` Registration page.

empty, and to `false` otherwise. The canonical representation abstracts out generic action commands and data values so that test steps in different contexts can be matched. Thus, the canonical representations of step 2 of t_1 and t_2 are the same: $[\![$`A`, `/html/body/table/.../tr/td/input`, `true`$]\!]$.

Next, we define path equivalence on the DOM. A DOM node has a type (*e.g.,* `div`, `table`, `button`, `image`) and a set of HTML attributes (*e.g.,* `id`, `name`, `class`, `src`) associated with it. Node n is *equivalent* to node m if and only if the nodes have the same type, the same set of attributes, and the same value for each attribute. Let $idx(n)$ denote the index of n. Let $p = (n_1, \ldots, n_k)$ and $q = (m_1, \ldots, m_k)$ be two paths in the DOM. Then, $p \equiv q$ if and only if, for all $1 \le i \le k$, n_i is equivalent to m_i and for all $1 \le i \le k-1$, $idx(n_i) = idx(m_i)$.

B. The Test-Modularization Algorithm

The algorithm has three phases, which we explain next.

1) Computation of Candidate Subroutines: Algorithm 1 presents the Phase 1 analysis; it takes as input a test suite T and produces as output the set of candidate subroutines for the tests in T. The algorithm partitions the steps of each test case based on the DOM locations of the referenced UI elements. The intuition behind this is that the proximity of the referenced UI elements indicates logical groupings of steps: test steps that access closely located elements in the DOM pertain to actions performed on a "page" or "form," whereas a step that accesses a significantly different part of the DOM than its preceding step potentially begins a new subroutine. More concretely, the analysis determines this by computing the least common ancestor (LCA) of a set E of UI element nodes. The farther away that the LCA of E is from the DOM root node, the more likely that the UI elements in E are related. Conversely, if the LCA is close to the root, E contains widely dispersed, and potentially unrelated, UI elements.

A *candidate subroutine* $S_c = (D, \psi, E, p)$, where D is a DOM, ψ is a sequence of test steps, E is the set of UI elements referenced in the steps in ψ, and p, referred to as *path prefix*, is the path in D from the root node to the LCA of all nodes in E. Consider Figure 3. Suppose that E consists of nodes 14 (the Login text box), 16 (the Password text box), and 19 (the Register button); these nodes are highlighted in Figure 3. Then, the LCA of these nodes is node 8 and p is the path $(1, 3, 8)$, shown as the dashed path in Figure 3.

As mentioned earlier, Phase 1 can be performed statically or dynamically; Algorithm 1 presents the dynamic variant of

Algorithm 1: Identification of candidate subroutines.

```
Input: Test suite $T$
Output: Set $\mathcal{S}_c$ of candidate subroutines
1  foreach $t \in T$ do
2      initialize $S_c = (D, \psi, E, p), l_{prev}$
3      foreach $s = (a, e, d) \in t$ do
4          let $D_{curr}$ be the DOM snapshot before step $s$
5          let $l_{curr}$ be the LCA of $e$ and $l_{prev}$ in $D_{curr}$
6          if startNewSubroutine($l_{prev}, l_{curr}, D_{curr}, E$) then
7              let $l$ be the LCA in $D_{curr}$ of the elements in $E$
8              set $D$ to the subtree in $D_{curr}$ rooted at $l$
9              set $p$ to the path from the root node to $l$ in $D_{curr}$
10             add $S_c$ to $\mathcal{S}_c$
11             reinitialize $S_c, l_{prev}$
12         else
13             set $l_{prev}$ to $l_{curr}$
14         add $e$ to $E$, add $s$ to $\psi$
15 return $\mathcal{S}_c$
```

Phase 1, which assumes that an execution trace containing the DOM snapshot before each test step is available. After explaining the dynamic variant of the algorithm, we discuss the modifications required for the static variant.

The algorithm iterates over each test case t (line 1) and each step in t (line 3). Line 4 reads the DOM snapshot before step s from the execution trace, and line 5 computes the LCA of e (the UI element referenced at s) and l_{prev} (the LCA from the previous iteration, which is undefined for the first test step). The LCA computation locates the nodes for e and l_{prev} in D_{curr}. Both e and l_{prev} have XPaths associated with them, so the algorithm uses path equivalence to locate them in D_{curr}. If step s results in the loading of a new page, l_{prev} would not appear in D_{curr}; in this case, e is used as the LCA in line 5.

Next, the algorithm calls the `startNewSubroutine()` function (line 6), which uses two metrics to determine whether a new subroutine should be started at s.

First, `startNewSubroutine()` checks whether all elements in E (*i.e.,* the elements referenced by steps in the candidate subroutine under consideration) appear in D_{curr}. If this is the case, the function proceeds with the second check; otherwise, it returns true, indicating that a new subroutine should be started. Typically, the latter condition would indicate a server-side communication that re-renders the DOM, causing previously referenced elements to no longer be available. (This check is performed only in the dynamic variant of Phase 1.)

Second, `startNewSubroutine()` checks whether the normalized difference between the distances of l_{prev} and l_{curr} from the root node of D_{curr} exceeds a threshold value, and returns true if it does. The check is performed as

$$\frac{\delta(l_{prev}, D_{curr}) - \delta(l_{curr}, D_{curr})}{\delta(l_{prev}, D_{curr})} > \lambda_d$$

where $\delta(n, D)$ returns the distance of n from the root node of D. Intuitively, this formula captures whether s accesses a UI element in a substantially different part of the DOM than the part accessed in the preceding steps, thus causing a big shift in the LCA node toward the DOM root. To accommodate cases where l_{prev} is close to the root, before applying the distance metric, `startNewSubroutine()` checks whether l_{curr} is the HTML `body` element and, if it is, returns true.

TABLE I
CANDIDATE SUBROUTINES COMPUTED FOR t_1 AND t_2 IN PHASE 1.

Sub-routine	D	ψ	E	p
$S^1_{c_1}$	DOM rooted at node 5	$\langle 1 \rangle$	$\{5\}$	$(1, 2, 5)$
$S^1_{c_2}$	DOM rooted at node 8	$\langle 2, 3, 4, 5, 6, 7 \rangle$	$\{14, 16, 18, 19, \ldots\}$	$(1, 3, 8)$
$S^1_{c_3}$	DOM rooted at node 6	$\langle 8 \rangle$	$\{6\}$	$(1, 2, 6)$
$S^1_{c_4}$	...	$\langle 9, 10, 11, 12 \rangle$	$\{\ldots\}$	$(\ldots)$
$S^2_{c_1}$	DOM rooted at node 5	$\langle 1 \rangle$	$\{5\}$	$(1, 2, 5)$
$S^2_{c_2}$	DOM rooted at node 8	$\langle 2, 3, 4, 5, 6, 7 \rangle$	$\{14, 16, 19, \ldots\}$	$(1, 3, 8)$

If a new subroutine is to be started, the algorithm updates the information for the current subroutine S_c (lines 7–9), adds S_c to the set of candidate subroutines (line 10), and re-initializes S_c and l_{prev} (line 11). However, if `startNewSubroutine()` returns false, the algorithm simply updates l_{prev} for the next iteration. In either case, it also adds e and s to S_c.

Consider the partitioning of the steps of t_1 (Figure 1) using the DOM shown in Figure 3. The first test step accesses the Registration image, which corresponds to node 5 in the DOM. Line 5 of Algorithm 1 sets l_{curr} to node 5 (l_{prev} is undefined). In the next iteration, test step 2, which accesses the Login text box (node 14), is processed. The LCA of nodes 5 and 14 is the root node. The distance metric on the old LCA (node 5) and the new LCA (node 1) evaluates to 1 ($\delta(l_{curr}, D_{curr}) = 0$), which exceeds the threshold λ_d (setting $\lambda_d = 0.75$, the value we used in our experiments). Thus, the processing of the current candidate subroutine is complete—it contains step 1 only—and step 2 and node 14 are added to a new candidate subroutine. Next, test step 3 is processed for which the accessed UI element is node 16. The new LCA is node 8, and the distance metric is calculated as 0.5, which is less than λ_d. Therefore, step 3 and node 16 are added to the current subroutine. Continuing in this way, the algorithm adds steps 4–7, for which the referenced UI elements occur in the subtree rooted at node 8, to the current subroutine. After step 7, a new page is loaded on which step 8 is performed; thus, a new subroutine is initialized.

In the end, the algorithm computes four candidate subroutines for t_1, shown in rows 1–4 of Table I. Two of the subroutines, $S^1_{c_1}$ and $S^1_{c_3}$, have only one test step each, whereas the remaining two, $S^1_{c_2}$ and $S^1_{c_4}$, have six and four steps, respectively. Similarly, two candidate subroutines are computed for t_2, shown in the last two rows of Table I.

The static variant of Phase 1 differs from the dynamic variant in two ways. First, at line 4, D_{curr} is not set to the DOM snapshot; instead, D_{curr} is built incrementally by adding XPaths to it for the referenced UI elements, one at a time. Thus, D_{curr} corresponds to the DOM *induced* by the test steps in the current candidate subroutine. Second, in the static variant, `startNewSubroutine()` does not perform the first check—that check is redundant because all elements in E always occur in the induced DOM. The dynamic variant can be more accurate in computing subroutines than the static variant, but it comes with the cost of collecting execution traces. In one of the empirical studies, we compared the effectiveness of the dynamic and static variants of Phase 1 (Section III-C).

2) Computation of Final Subroutines: Phase 2 starts by grouping the candidate subroutines into subroutine groups. It then refines each subgroup, possibly splitting a subgroup into smaller subgroups, to compute the final set of subroutines.

A *subroutine group* $S_G = \{S_{c_1}, \ldots, S_{c_k}\}, k \geq 1$, is a set of candidate subroutines such that for each pair $S_{c_i}, S_{c_j} \in SG, i \neq j, p_i \equiv p_j$. In other words, S_G is the set of candidate subroutines that have equivalent path prefixes. For the candidate subgroups for our running example (Table I), the technique computes four subroutine groups: $S_{G_1} = \{S^1_{c_1}, S^2_{c_1}\}$ groups two of the subroutines based on the equivalent path prefix $(1, 2, 5)$; $S_{G_2} = \{S^1_{c_2}, S^2_{c_2}\}$ groups two subroutines based on the equivalent path prefix $(1, 3, 8)$; the remaining two groups contain one candidate subroutine each.

These subroutine groups are a conservative initial grouping; the rest of Phase 2 refines the initial groups. First, it analyzes the overlap of UI elements among the subroutines in a group to refine the group into smaller groups. Second, it computes a total order of the test steps for a group, splitting a group when necessary to resolve sequencing conflicts.

Algorithm 2 presents this analysis; it takes as input a set $\mathcal{S}_{\mathcal{G}}$ of subroutine groups and returns the set $\mathcal{S}_f$ of final subroutines. Line 1–11 refine the groups based on element overlap. Intuitively, the analysis determines the "cohesion" of candidate subroutines in a group based on commonality of the referenced UI elements. In doing this, it serves to separate out potentially incorrect groupings: for example, two subroutines that perform actions on different pages, but coincidentally have the same path prefix (because of which they were grouped together) can be separated based on low element overlap. The analysis thus attempts to follow the reasoning that would be done in identifying page-object abstractions manually.

The algorithm iterates over each S_{G_i} in the input set of subroutine groups (line 2); it removes subroutines from S_{G_i} until S_{G_i} becomes empty (lines 3–10), creating one or more refined groups S_{G_o} in the process. It first sorts the subroutines in S_{G_i} based on the sizes of their element sets and initializes the output group S_{G_o} with the first subroutine (lines 4–5). Then, it incrementally builds S_{G_o} by moving those elements that have a high overlap with the set of DOMs in S_{G_o}. The parameter λ_e determines whether a candidate subroutine is moved to S_{G_o}. When the loop in line 6 terminates, some or all of the subroutines in S_{G_i} have been moved to S_{G_o}. If there are remaining subroutines in S_{G_i}, this process is repeated.

Note that the element intersection in line 8 is computed with respect to the DOM objects in the candidate subroutines. In the dynamic variant of Phase 1, even elements that are not referenced in a test case, but that occur in the DOM rooted at the LCA, appear in the subroutine DOM object. This has the nice property of accommodating the scenario where two tests navigate to the same form but access different sets of UI elements in the form, with little or no overlap. The element intersection of these tests, as computed in line 8, would still be high because the respective DOMs contain the non-referenced elements also. Thus, the initial grouping of these tests is maintained. However, for the static variant of

Algorithm 2: Computation of final subroutines.

```
   Input: Set $\mathcal{S}_\mathcal{G} = \{S_{G_1}, \ldots, S_{G_m}\}$ of subroutine groups
   Output: Set $\mathcal{S}_f = \{S_{f_1}, \ldots, S_{f_n}\}$ of final subroutines
   // Refine $\mathcal{S}_\mathcal{G}$ based on element overlap to create $\mathcal{S}_{\mathcal{G}1}$
1  initialize $\mathcal{S}_{\mathcal{G}1}$ to the empty set
2  foreach $S_{G_i} \in \mathcal{S}_\mathcal{G}$ do
3      while $S_{G_i} \neq \emptyset$ do
4          sort $S_{G_i}$ in decreasing size of element set $E$, initialize $S_{G_o}$
5          remove the first subroutine from $S_{G_i}$ and add it to $S_{G_o}$
6          foreach $S_c \in S_{G_i}$ do
7              let $E$ be the element set of $S_c$
8              let $E_s$ be elements of $E$ that occur in a DOM in $S_{G_o}$
9              if $(|E_s|/|E|) > \lambda_e$ then
10                 remove $S_c$ from $S_{G_i}$ and add it to $S_{G_o}$
11         add $S_{G_o}$ to $\mathcal{S}_{\mathcal{G}1}$
   // Compute final subroutines from $\mathcal{S}_{\mathcal{G}1}$, further refining the subroutine groups in
      case of conflicts in step sequences
12 initialize $\mathcal{S}_f$ to the empty set
13 foreach $S_{G_i} \in \mathcal{S}_{\mathcal{G}1}$ do
14     sort $S_{G_i}$ in decreasing length of step sequence
15     while $S_{G_i} \neq \emptyset$ do
16         remove the first sub. $S_{c_1}$ from $S_{G_i}$ and initialize $S_f$ with it
17         let $\psi_1$ be the step sequence of $S_{c_1}$
18         initialize graph $G$ with canonical representation of steps in $\psi_1$
19         foreach $S_c \in S_{G_i}$ do
20             let $\psi_c$ be the sequence of canonical rep. of the steps in $S_c$
21             if adding $\psi_c$ to $G$ does not create a cycle in $G$ then
22                 remove $S_c$ from $S_{G_i}$
23                 update test mapping information $\gamma$ for $G$
24         let $\Psi$ be the step sequence in topological sort order of $G$
25         add $(\Psi, \gamma)$ to $\mathcal{S}_f$
26 return $\mathcal{S}_f$
```

Phase 1—in which case the subroutine DOM is the *induced* DOM—Algorithm 2 would separate these tests into different subroutine groups because of low element overlap.

After refining $\mathcal{S}_\mathcal{G}$ into $\mathcal{S}_{\mathcal{G}1}$, Algorithm 2 computes the final subroutines (lines 12–25), which involves creating, for each subroutine group S_G in $\mathcal{S}_{\mathcal{G}1}$, a total order of step sequences.

Let $S_G = \{S_{c_1}, \ldots, S_{c_k}\}$ be a subroutine group. Let $\psi_1, \ldots, \psi_k$ be the step sequences in the subroutines in S_G, with each test step in its canonical representation. A *final subroutine* $S_f = (\Psi, \gamma)$ is constructed from S_G, where Ψ is an interleaving of $\psi_1, \ldots, \psi_k$ such that, for any ψ_i, the order of steps in ψ_i is preserved in Ψ; γ is a function that maps a step to the test cases in which the step occurs.

The total order in a final subroutine must ensure that the order of execution of steps in the original tests is preserved when the subroutine is called from those tests. This is necessary to guarantee behavior-preserving subroutine extraction. Consider the subroutine group S_{G_2} for the `Bookstore` example, which consists of two candidate subroutines (rows 2 and 6 of Table I) with six steps each, five of which are common: steps 2, 3, 5–7 of $S^1_{c_2}$ which come from test t_1 and steps 2–6 of $S^2_{c_2}$ which come from t_2. The total ordering shown in the extracted subroutine in Figure 2 preserves the execution order of the steps from t_1 and t_2.

In general, some "precedes" relations between test steps must be preserved (*e.g.,* steps 2–5 must execute before step 6 in t_2), whereas other relations can be relaxed (*e.g.,* the order of execution of steps 4 and 5 does not matter). But, without knowing the semantics of each step, it is not possible to determine which precedes relations can be relaxed. Thus, although steps could be reordered in a semantics-preserving manner, we take the conservative approach of ensuring that all precedes relations are preserved in the final subroutine; if conflicts occur, the subroutine group is refined via splitting.

The problem of computing a precedence-preserving total order, and detecting conflicts, can be solved by constructing a directed graph in which a cycle indicates conflicts, and a topological sort on the final (acyclic) graph gives a precedence-preserving total order. At a high level, this is what Algorithm 2 does in lines 13–25. It iterates over each subroutine group (line 13), and processes each candidate subroutine, starting with the one with the longest step sequence (lines 14–15). Line 18 initializes a graph G in which nodes represent canonical representations of test steps and edges represent the order of execution among steps. (The graph can have multiple initial and final nodes.) Then, for each subroutine S_c in the group, the algorithm attempts to add the steps of S_c, in order, to G. If the addition of a step and its outedge results in a cycle in G, the algorithm has detected a conflicting sequence. In that case, it omits S_c and proceeds to the next subroutine. If the steps in S_c cause no cycles, it is removed from S_{G_i} and the step-to-tests map γ is updated (lines 22–23) with the newly added steps from S_c.

After each subroutine has been processed, a topological sort of G gives the total step sequence Ψ, which along with the map γ, is added to the set of final subroutines (lines 24–25).

3) Generation of Refactored Test Suite: Phase 3 uses the set of final subroutine to create the transformed test suite. This phase creates a callable subroutine for each final subroutine $S_f = (\Psi, \gamma)$ in $\mathcal{S}_f$. To do this, it first determines which statements in Ψ execute conditionally and which ones execute unconditionally (in other words, in all calling contexts). The mapping information γ maps each test step in Ψ to the set of tests containing that step. Thus, the union of the tests in the range of γ gives the set of tests T_c that forms all calling contexts for S_f. Then, for any step $s \in \Psi$, if $\gamma(s) = T_c$, s executes unconditionally; otherwise, it executes conditionally. The algorithm creates a formal control parameter and encloses the conditionally-executing test steps of the subroutine in `if` statements, as illustrated for `EnterRegDetails` in Figure 2. Also, for each test step that takes a data value, the data reference is parameterized.

After creating the subroutines, the algorithm refactors each test case by replacing tests steps with calls to the subroutines, and adding statements that set up the actual parameters for the calls with context-specific data and control values.

III. Empirical Evaluation

We implemented our technique and conducted three empirical studies using open-source web applications to evaluate various aspects of the technique. After describing the experimental setup, we present the results of the studies.

A. Experimental Setup

We implemented the static and dynamic variants of our technique as an extension to ATA [24], which is a tool for

TABLE II
SUBJECTS USED IN THE EMPIRICAL STUDIES.

Subject	Description	Tests	Test Steps Total	Avg	Min	Max
Bookstore	Shopping portal for books	57	643	11	7	28
Classifieds	Portal for posting and checking advertisements	30	332	11	6	21
jBilling	Enterprise billing application	73	748	10	5	18
Tudulist	Portal for managing personal TODOs	19	146	7	6	10
Total		179	1869			

creating GUI tests for web applications. The implementation analyzes the tests created in ATA: given a suite of test cases, it creates a modularized test suite, consisting of Selenium [1] test cases coded in Java, that can be executed using JUnit. The modularized test suite consists of parameterized subroutines, each of which is a static Java method, defined in a `Modules` class, that takes an array of strings (data values) and an array of booleans (control predicates) as formal parameters. A method is parameterized only if it contains at least one test step that requires a data value or that needs to execute conditionally. Each transformed ATA test case is emitted as a JUnit test case.

We also implemented a trace-collection feature, which records the complete browser DOM before the execution of each test step; this is done using the Selenium API.

We used four open-source web applications as experimental subjects; Table II lists the applications. Among these, `jBilling` and `Tudulist` are AJAX applications, whereas the others are JSP applications.

Table II also presents information about the test cases for the subjects. We created test suites consisting of functional test cases. We designed the test cases to cover various application features. For example, the 57 test cases for `Bookstore` exercise application functionality such as adding, modifying, or deleting members, books, orders, etc. The tests cover normal scenarios and error scenarios (*e.g.,* attempting to register with an existing user ID). Over all subjects, we created 179 functional tests cases, consisting of 1869 test steps. The average number of test steps ranges from 7 to 11; the largest test, consisting of 28 steps, occurs in the test suite for `Bookstore`. (The evaluation dataset is available at: sites.google.com/site/irlexternal/test-modularization.)

We automated the test cases using ATA [24], and then executed the dynamic and static variants of the implementation to generate modularized Selenium test cases. As a sanity check for correctness, we executed each refactored test to ensure that it passed. At the minimum, this guarantees that subroutine extraction does not alter the (passing) outcome of test cases.

In the first study, we used the dynamic variant of the technique (which, in general, is more effective than the static variant), and the parameter values $\lambda_d = 0.75$ and $\lambda_e = 0.8$. In the second study, we compared the static and dynamic variants (using the default parameter values above). In the final study, we evaluated the effects of varying λ_d and λ_e.

B. Study 1: Effectiveness

Goals and Method: In the first study, we evaluated the effectiveness of our technique. We collected data about the

TABLE III
SIZES OF THE REFACTORED TESTS AND EXTRACTED SUBROUTINES.

Subject	Tests	Refactored Test Steps Tot	Avg	Min	Max	Sub-routines	Subroutine Steps Tot	Avg	Min	Max	Call Sites
Bookstore	57	223	3	1	13	19	104	5	2	13	130
Classifieds	30	103	3	0	9	12	67	5	2	21	69
jBilling	73	107	1	0	3	16	79	4	2	11	158
Tudulist	19	5	0	0	1	3	33	11	3	22	33
Total	179	438				50	283				390

reduced lengths of the refactored tests and data about the extracted subroutines, such as their lengths and number of calling contexts. Moreover, to quantify the simplification attained in the refactored test suite (T_r) over the original test suite (T), we used three metrics.

The first metric, *reduction index* (RI), quantifies the decrease in the number of test steps in T_r, whereas the second metric, *call index* (CI), quantifies the reuse via calls to subroutines, for a refactored test suite:

$$RI = 1 - \frac{\sum\limits_{t \in T_r} st(t) + \sum\limits_{S \in \mathcal{S}_f} st(S)}{\sum\limits_{t \in T} st(t)} \qquad CI = 1 - \frac{|\mathcal{S}_f|}{\sum\limits_{S \in \mathcal{S}_f} cs(S)}$$

where $st(t)$ is the number of steps, excluding calls, in test t, $st(S)$ is the number of steps in subroutine S, and $cs(S)$ is the number of calls to S. Both RI and CI range from zero (inclusive) to one (exclusive), with higher values indicating greater reduction in test steps and greater reuse, respectively.

Although high reduction via extraction of highly reusable subroutines is desirable, the complexity of the extracted subroutines—in terms of the distinct flows that occur through them—can make the refactored tests harder to understand, thereby negatively impacting the simplification achieved. To investigate this aspect of modularization, we define the number of *distinct behaviors* of a subroutine as the distinct sequences of steps, abstracting out the data values, that execute over all calling contexts of the subroutine.[2] Our third metric, subroutine *behavior index* (BI), quantifies this for a refactored test suite:

$$BI = \frac{|\mathcal{S}_f|}{\sum\limits_{S_f \in \mathcal{S}_f} db(S_f)}$$

where $db(S_f)$ is the number of distinct behaviors of subroutine S_f. BI varies from zero (exclusive) to one (inclusive), with higher values indicating fewer distinct behaviors, which is desirable from the perspective of comprehension complexity.

Results and Analysis: Table III presents the basic data about the sizes of the refactored tests and the extracted subroutines. Columns 3–6 show data about the refactored test cases. A refactored test case can contain UI actions and calls to the extracted subroutines: in measuring the size of a refactored test, we count only the UI actions. As the data illustrate, the tests become smaller after subroutine extraction: the total number of test steps decreases from 1869 to 438 (77% reduction), whereas the average test size over all subjects, decreases from

[2]Note that, by our definition, distinct behaviors are a subset of the linearly independent paths in the control-flow graph of a subroutine [25]: they represent only those paths that execute in at least one calling context of the subroutine.

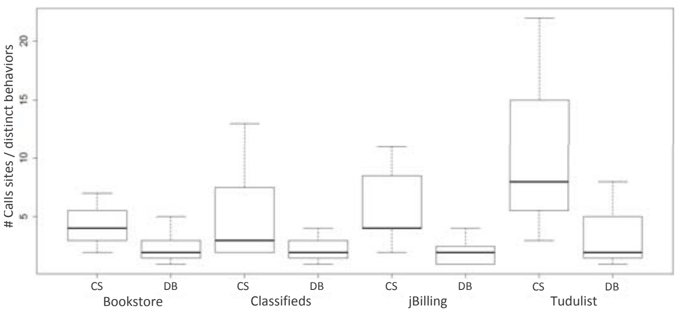


Fig. 4. Distribution of call sites and distinct behaviors for extracted subroutines; "CS" represents call sites, whereas "DB" represents distinct behaviors.

9.75 to 1.75. In three of the subjects, the smallest test case ends up containing only subroutine calls (and no UI actions), indicated by the "Min" values of zero.

Columns 7–11 show data about the extracted subroutines: 50 subroutines are extracted over the four subjects, containing a total of 283 steps (or UI actions). On average, the subroutines contain about six steps; the largest subroutine, which occurs in `Tudulist`, has 22 steps. Thus, the technique is able to extract reasonably sized subroutines, which are, on average, more than half as long as the original tests.

The next question about the extracted subroutines is about their reuse: how many times are the subroutines called (and thus reused) by test cases? The last column of Table III lists the total number of call sites created in the refactored tests. For example, for the 19 subroutines extracted for `Bookstore`, 130 call sites are created—on average, over six call sites per subroutine. For `jBilling`, the average is over nine call sites per subroutine. To provide further details on call-site distribution, Figure 4 presents the data for all subroutines using a boxplot (plots labeled "CS"), which shows median value, the range of the quartiles, and the outliers.[3]

Each subroutine has at least two call sites, which is ensured by our algorithm: it does not extract subroutines that have only one caller. For `Bookstore`,`jBilling`, and `Tudulist` 50% or more of the subroutines have four or more calls to them. Excluding the data points not plotted,[2] the maximum numbers of call sites for any subroutine in `Bookstore`, `Classifieds`, `jBilling` and `Tudulist` are 10, 21, 16 and 22 respectively.

Listing 1 shows an example extracted subroutine for `jBilling`. For `jBilling`, we have test cases that exercise the functionality of adding a new partner. This requires the tests to navigate to a registration page, in which various details, such as login name, password, email ID, etc., have to be entered. The test cases either save the changes on the page or discard the changes by clicking Cancel. These steps are extracted into a subroutine, as shown in the top part of Listing 1; the bottom part shows call sites from two test cases that invoke the subroutine with appropriate data and control parameters. The illustration in Listing 1 is a simplified version of this subroutine; the actual subroutine incorporates more steps and is invoked from eight call sites.

[3]The data is plotted after eliminating three data points: one subroutine each in `Bookstore`, `Classifieds`, and `jBilling`, with 54, 21, 73 call sites, respectively. In each case, these are "login" subroutines that are called from many test cases—in some cases at the beginning of test cases and, in other cases, in the middle of test flows.

```
public static void enter_login_name_password_verify_password
 (String[] data, boolean[] control) {
   execute("enter","Login Name",data[0],locator,false);
   execute("enter","Password",data[1],locator,false);
   execute("enter","Verify Password",data[2],locator,false);
   execute("enter","Next payout date",data[3],locator,false);
   execute("select","Related clerk",data[4],locator,false);
   execute("enter","email",data[5],locator,false);
   if (control[0])
      execute("click", "save changes", "", locator, false);
   if (control[1])
      execute("click", "cancel", "", locator, false);
}
```

```
// Test Case 1: Save changes after entering details
enter_login_name_password_verify_password(new String[]
{"partner1", "Pass123$", "Pass123$", "06/06/2014", "clerk1",
"partner1@xyz.com"}, new boolean[] {true, false});

// Test Case 2: Cancel after entering details
enter_login_name_password_verify_password(new String[]
{"partner1", "Pass123$", "Pass123$", "06/06/2014", "clerk1",
"partner1@xyz.com"}, new boolean[] {false, true});
```

Listing 1. An extracted subroutine and two related call sites for `jBilling`.

TABLE IV
SIMPLIFICATION ACHIEVED IN THE REFACTORED TEST SUITES.

Subject	Reduction Index (RI)	Call Index (CI)	Behavior Index (BI)
Bookstore	0.49	0.85	0.42
Classifieds	0.49	0.83	0.36
jBilling	0.75	0.90	0.46
Tudulist	0.74	0.91	0.27

We manually examined the subroutines for `Bookstore`, `Classifieds`, and `Tudulist` to see whether they correspond to pages or forms in those applications. For `Classifieds`, 11 of the 12 subroutines roughly correspond to an application page, whereas one subroutine combines elements from two pages—Member Record and Registration—that have similar forms pertaining to member information. We found three similar instances in `Bookstore`; the remaining 16 subroutines correspond to a form. For `Tudulist`, two of the subroutines correspond to Login and Registration features, each of which is implemented on one page; the third subroutine corresponds to the main page, which supports different features, such as adding, deleting, or editing lists, all of which cause the same page (with the list of available items) to be displayed.

Table IV shows the data about the reduction, call, and behavior indexes. As the data show, a significant amount of reduction, or removal of duplication, occurs for `jBilling` and `Tudulist`, where the total number of UI action steps in the refactored suites is less than 30% of what they were in the original test suites. For `Bookstore` and `Classifieds`, the reduction is less, but still close to 50%. The CI values are consistently high for all subjects, indicating that the extracted subroutines have a very high degree of reuse via calls from multiple call sites. The behavior index ranges from 0.27 to 0.46. On average, the subroutines for `Tudulist` exhibit about four distinct behaviors, whereas the subroutines for `jBilling` exhibit only two distinct behaviors.

Figure 4 also shows the distribution of distinct behaviors of subroutines (plots labeled "DB"). In all cases, the number of distinct behaviors is significantly lower than the number of callers, indicating that multiple call sites invoke a subroutine

TABLE V
COMPARISON OF THE DYNAMIC AND STATIC VARIANTS.

	Dynamic Variant						Static Variant					
Subject	# Test Steps	# Sub	Avg Sub Size	*RI*	*CI*	*BI*	# Test Steps	# Sub	Avg Sub Size	*RI*	*CI*	*BI*
Bookstore	223	19	5	0.49	0.85	0.42	279	15	5	0.44	0.84	0.58
Classifieds	103	12	5	0.49	0.83	0.36	161	7	12	0.26	0.65	0.41
jBilling	107	16	4	0.75	0.90	0.46	47	13	11	0.73	0.81	0.37
Tudulist	5	3	11	0.74	0.91	0.27	19	5	8	0.71	0.74	0.50
Tot / Avg	438	50	6.25	0.62	0.87	0.38	506	40	9	0.54	0.76	0.47

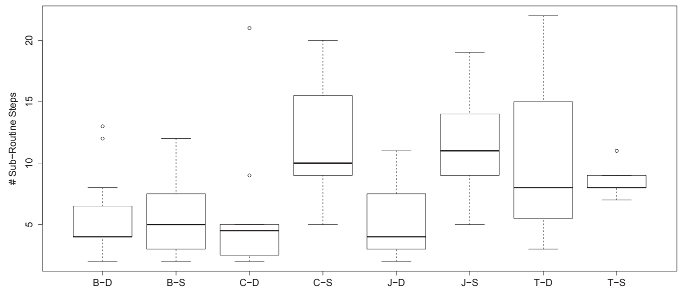


Fig. 5. Distribution of subroutine sizes for the dynamic and static variants. The subject names are abbreviated to their initials; "D" indicates dynamic, whereas "S" represents the static variant.

with the same control parameters. For `Classifieds` and `jBilling`, none of the subroutines have more than four distinct behaviors; for `Bookstore`, none of the subroutines have more than five distinct behaviors, with 75% of the subroutines having at most three distinct behaviors. Even in the case of `Tudulist`, for which the subroutines exhibit relatively more behaviors, most of the subroutines have five or less distinct behaviors; for the remaining, the number of behaviors ranges between six and eight.

C. Study 2: Comparison of Dynamic and Static Variants

Goals and Method: In the second study, we compared the dynamic and static variants. In general, we expect the dynamic variant to be more effective in extracting subroutines that have high reduction and reuse, with low distinct behaviors. We collected data to compare the variants in terms of the sizes of the refactored tests, the number of subroutines, the average subroutine size, the three indexes, and execution times.

Results and Analysis: Table V presents the data comparing the variants on these metrics. The dynamic variant identifies more subroutines (50 versus 40) that have smaller lengths on average (6.25 steps versus 9 steps) than the static variant. The dynamic variant also outperforms the static variant on the reduction index and the call index for all subjects. On the BI metric, however, the results are mixed and more favorable to the static variant. While the dynamic variant is more effective for `jBilling`, it is less effective for the other subjects. This is particularly notable for `Tudulist`, where the subroutines for the dynamic variants can have twice as many distinct behaviors (four versus two) as the subroutines computed by the static variant. For `Tudulist`, the dynamic variant computes a large subroutine (mentioned earlier), which contains 22 steps and has 33 callers and 11 distinct behaviors.

Figure 5 presents the distribution of the subroutine sizes for the dynamic and static variants. With the exception of

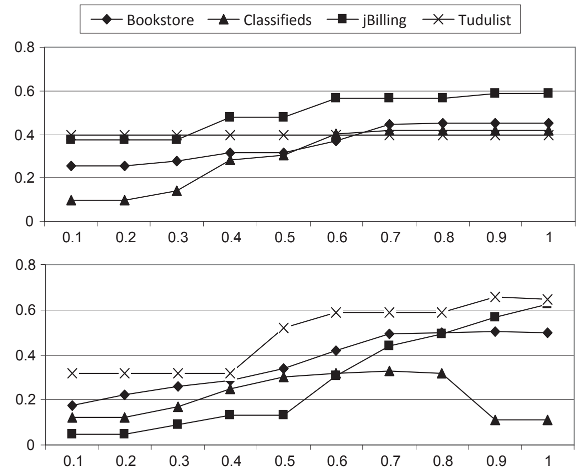


Fig. 6. Effects of varying λ_e (horizontal axis) on the harmonic mean of RI, CI, and BI for the dynamic variant (top) and the static variant (bottom).

`Tudulist`, the dynamic variant produces smaller subroutines than the static variant, and this difference is quite significant for `Classifieds` and `jBilling`.

In terms of execution performance, the static variant executed in about a second or less for each subject. The dynamic variant has a large overhead for loading execution traces; after the traces are loaded, the time for modularaization is roughly the same as that for the static variant. For `jBilling`, the execution time was a little over a minute (most of which was spent in loading a 60MB trace file), whereas for the other subjects, the dynamic variant took about 10 seconds or less.

D. Study 3: Effects of the Configuration Parameters

Goals and Method: In the final study, we investigated the effects of varying the LCA distance parameter λ_d (Algorithm 1) and the element-overlap parameter λ_e (Algorithm 2). We varied λ_d and λ_e from 0.1 to 1 in increments of 0.1 for both the dynamic and static variants. As the dependent variable, we used the harmonic mean of RI, CI, and BI, which we refer to as the *modularization index* (MI). The harmonic mean is used as it tends to privilege balanced systems compared to other means such as the arithmetic mean [26].

Results and Analysis: For λ_d, with the dynamic algorithm, we did not see any difference in MI, whereas with the static variant, MI decreased slightly with increase in λ_d, before stabilizing. We next discuss in more detail the results for λ_e for which we observed greater variations in MI.

Figure 6 presents the plot of the modularization index against different values of λ_e for the dynamic and static variants. A general trend that can be observed is that the MI values increase monotonically and plateau out around 0.7, with only minor or no increases after that. There are a couple of notable exceptions in the static case: MI continues to increase for `jBilling` even for $\lambda_e > 0.7$ and decreases sharply for `Bookstore` for $\lambda_e > 0.8$. Based on this data, the range 0.7–0.8 for λ_e seems to offer the best trade-off between accommodating candidate subroutines with some diversity to be merged, while also attaining high scores for the reduction, call, and behavior indexes.

More generally, the static variant displays a greater sensitivity to λ_e, and therefore greater instability, than the dynamic variant, which also suggests that the dynamic algorithm is superior to the static algorithm for subroutine extraction.

E. Discussion

Although our results are promising, like any empirical study, there are threats to the validity of our results. The most significant of these are threats to external validity, which arise when the observed results cannot be generalized to other experimental setups. In our study, we used four subjects with manually created functional test suites. The performance of the technique may vary for other subjects and types of test suites (*e.g.,* tests generated via automated techniques, with different coverage goals).

IV. Related Work

There is a large body of work on automated techniques for generating GUI tests, based on different notions of coverage, in the context of web applications (*e.g.,* [12]–[16], [19], [20]) and other forms of GUIs (*e.g.,* [17], [18], [21], [27]). The tests generated by such techniques are essentially sequences of GUI events (*e.g.,* as illustrated in Figure 1(b)); thus, being non-modular, they can be hard to maintain when intended for use in regression testing. This has motivated the development of techniques for automatically repairing test cases (*e.g.,* [5]–[8], [28]), or generating less-brittle tests upfront or tests with reliable oracles (*e.g.,* [4], [29]) that require less maintenance. An orthogonal approach toward simplifying maintenance is to modularize the tests, but this has not received much attention in the testing research community.

The work that comes closest to ours in terms of the objective is the test-modularization approach presented by Mahmud and Lau [22]. Their technique uses supervised machine learning (requiring a training sample of subroutines) and analyzes tests written in ClearScript [23], [30]. Given a training set of subroutines, it computes similarity of test-case instruction sequences with instructions in the labeled subroutines, and uses heuristics to accommodate partial matches and generalizations. The effectiveness of that techniques, in general, depends on the diversity of the training set. Moreover, it does not analyze the referenced UI elements and the structure of the navigated webpages, which can affect its accuracy in two ways. First, two test steps that read the same in ClearScript may, in fact, refer to UI elements on different pages. Therefore, similar-looking actions on different pages could be erroneously marked as belonging to the same subroutine. Second, two sequences of instructions that reach the same page, but perform different actions on the page would have low similarity, but nonetheless, could be considered as part of the same subroutine (from the perspective of maintaining test scripts). In such cases, the technique could fail to identify reusable subroutines. Our technique addresses these issues by analyzing the UI elements and the structure of the pages accessed in the tests, and not relying on the availability of training samples.

In previous work, we presented a technique for merging similar GUI tests in a semantics-preserving manner [31]. The goal was to improve the *execution efficiency* of GUI tests by eliminating repeated executions of the same sequence of actions in the context of different test cases (while preserving the fault-detection effectiveness). Given a suite of GUI tests, the technique identifies the tests that can be combined and creates a merged test, which covers all the application states exercised individually by the tests, but with the redundant common steps executed only once. In contrast, the goal of this work is to improve the *maintenance efficiency* by extracting out reusable subroutines and making GUI tests modular. Although modularization can simplify maintenance, it does not improve execution efficiency—*e.g.,* the common steps of t_1 and t_2 (Figure 1) execute twice, in the contexts of those tests, after modularization.

In programming, *ExtractMethod* code refactoring refers to the extraction of code statements from a given method into another newly created method. The end goals of *ExtractMethod* code refactoring and our test-case modularization are the same—increased modularization and better maintainability. However, *ExtractMethod* code refactoring is distinct from test-case modularization in the techniques used to achieve the end goal. Method extraction is primarily based on program slicing [32], [33] to make the dependence-related statements contiguous for extraction. Our technique is applicable to GUI tests cases that consist of sequences of events on the application user interface; it extracts contiguous events into a subroutine with parameterized data and control flow.

V. Summary and Future Work

We presented a new technique for automated modularization of GUI test cases. Unlike learning-based GUI test modularization [22], our technique analyzes the references to UI elements in test steps and the induced DOM states, and requires no training samples of labeled subroutines. Our approach can be applied to GUI test cases created via manual recording or automated test-generation techniques (*e.g.,* [12], [13], [15], [19]), both of which result in non-modular tests. Efficiently modularizing such tests using an automated technique, such as ours, can reduce the maintenance cost of the tests and, thus, make them more suitable for use in regression testing. Our empirical results demonstrated the effectiveness of the technique in extracting highly reusable subroutines and reducing the sizes of test cases, while keeping the comprehension overhead of the subroutines low.

More experimentation with varied subjects and test suites (*e.g.,* considering tests generated by automated techniques) would help confirm the generality of our results. Our technique could be extended by iterative refinement of subroutines so that multi-level subroutine decomposition can be performed, instead of one-level decomposition which our current algorithm does. Such refinement could be guided, for example, by an upper bound on the number of distinct behaviors. Finally, future research could investigate how modular and non-modular tests compare in terms of maintenance and comprehension effort as the application under test evolves.

References

[1] "Selenium," http://seleniumhq.org/.

[2] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Capture-replay vs. programmable web testing: An empirical assessment during test case evolution," in *WCRE*, 2013, pp. 272–281.

[3] S. Thummalapenta, P. Devaki, S. Sinha, S. Chandra, S. Gnanasundaram, D. D. Nagaraj, and S. Sathishkumar, "Efficient and change-resilient test automation: An industrial case study," in *ICSE Software Engineering in Practice Track*, 2013, pp. 1002–1011.

[4] R. Yandrapally, S. Thummalapenta, S. Sinha, and S. Chandra, "Robust test automation using contextual clues," in *ISSTA*, 2014, pp. 304–314.

[5] A. M. Memon, "Automatically repairing event sequence-based GUI test suites for regression testing," *ACM Transaction Software Engineering and Methodology*, vol. 18, pp. 1–36, November 2008.

[6] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and evolving GUI-directed test scripts," in *ICSE*, 2009, pp. 408–418.

[7] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "WATER: Web application test repair," in *ETSE*, 2011, pp. 24–29.

[8] S. Zhang, H. Lü, and M. D. Ernst, "Automatically repairing broken workflows for evolving GUI applications," in *ISSTA*, 2013, pp. 45–55.

[9] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro, "Improving test suites maintainability with the page object pattern: An industrial case study," in *ICST Workshops*, 2013, pp. 108–113.

[10] "Page Objects," https://code.google.com/p/selenium/wiki/PageObjects.

[11] A. M. Memon, "An event-flow model of GUI-based applications for testing," *Softw. Test., Verif. Reliab.*, vol. 17, no. 3, pp. 137–157, 2007.

[12] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of Ajax web applications," in *ICST*, 2008, pp. 121–130.

[13] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 35–53, Jan/Feb 2012.

[14] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "Rich internet application testing using execution trace data," in *ICST Workshops*, 2010, pp. 274–283.

[15] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra, "Guided test generation for web applications," in *ICSE*, 2013, pp. 162–171.

[16] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-based web applications through dynamic analysis of user interface state changes," *ACM Trans. on the Web*, vol. 6, no. 1, pp. 1–30, Mar. 2012.

[17] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: An innovative tool for automated testing of GUI-driven software," *Automated Software Engg.*, vol. 21, no. 1, pp. 65–105, Mar. 2014.

[18] F. Gross, G. Fraser, and A. Zeller, "EXSYST: Search-based GUI testing," in *ICSE*, 2012, pp. 1423–1426.

[19] A. M. Fard and A. Mesbah, "Feedback-directed exploration of web applications to derive test models," in *ISSRE*, 2013, pp. 278–287.

[20] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *ICSE*, 2011, pp. 561–570.

[21] S. Ganov, C. Killmar, S. Khurshid, and D. E. Perry, "Event listener analysis and symbolic execution for testing GUI applications," in *ICFEM*, 2009, pp. 69–87.

[22] J. Mahmud and T. Lau, "Lowering the barriers to website testing with CoTester," in *IUI*, 2010, pp. 169–178.

[23] G. Leshed, E. M. Haber, T. Matthews, and T. Lau, "Coscripter: Automating and sharing how-to knowledge in the enterprise," in *CHI*, 2009, pp. 1719–1728.

[24] S. Thummalapenta, N. Singhania, P. Devaki, S. Sinha, S. Chandra, A. K. Das, and S. Mangipudi, "Efficiently scripting change-resilient tests," in *FSE Tool Demonstrations Track*, 2012, pp. 41:1–41:2.

[25] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[26] D. Nadeau and S. Sekine, "A survey of named entity recognition and classification," *Lingvisticae Investigationes*, vol. 30, no. 1, pp. 3–26, 2007.

[27] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "AutoBlackTest: Automatic black-box testing of interactive applications," in *ICST*, 2012, pp. 81–90.

[28] S. Huang, M. B. Cohen, and A. M. Memon, "Repairing gui test suites using a genetic algorithm," in *ICST*, 2010, pp. 245–254.

[29] D. Roest, A. Mesbah, and A. van Deursen, "Regression testing Ajax applications: Coping with dynamism," in *ICST*, 2010, pp. 127–136.

[30] G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan, "Koala: capture, share, automate, personalize business processes on the web," in *CHI*, 2007, pp. 943–946.

[31] P. Devaki, S. Thummalapenta, N. Singhania, and S. Sinha, "Efficient and flexible GUI test execution via test merging," in *ISSTA*, 2013, pp. 34–44.

[32] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities," in *CSMR*, 2009, pp. 119–128.

[33] R. Komondoor and S. Horwitz, "Effective, automatic procedure extraction," in *IWPC*, 2003, pp. 33–42.