Dependency Detection and Repair in Web Application Tests

Ali M. Alakeel

Abstract—Various techniques have been created for manipulating test suites in order to improve the cost-effectiveness of regression testing; these include regression test selection, test prioritization, test execution automation, test factoring, and test carving techniques. Dependencies between tests can render these techniques problematic, by causing tests to break (i.e., execute inappropriately or fail to execute at all). For this reason, researchers have begun to seek approaches for detecting or eliminating test dependencies. To date, this work has focused primarily on detecting test dependency while leaving the tedious and labor-intensive work to repair the damage caused by these dependencies to be performed manually by the developer. This paper presents WebTestRepair, the first automated algorithm for test dependency detection and repair in web application test suites. WebTestRepair performs a combined string and data-flow analysis which focus on identifying manifest test dependencies that cause test failures and then automatically repair the damage caused by these dependencies. As a result of this repair process a previously broken test suite is now able to execute without test failures. An empirical study to evaluate WebTestRepair on five open-source web applications without test failures. An empirical study to evaluate WebTestRepair on five open-source web applications shows that WebTestRepair was able to identify and repair manifest test dependencies in 95% of the scenarios in which they occurred and that it was also over 81% effective than a technique that randomly re-orders tests in an attempt to repair test breakages.

Index Terms—test dependencies, web applications testing, automated Algorithms, software engineering.

I. INTRODUCTION

SOFTWARE engineers create test suites in order to locate faults in software systems. Later, test suites are used in regression testing to determine whether subsequent versions of those systems behave appropriately. At regression testing time, engineers may seek to increase the cost-effectiveness of testing by applying some sort of test manipulation technique to their test suites, such as test prioritization (TCP) techniques (e.g., [11, 19, 20], [25, 36]) or regression test selection (RTS) techniques (e.g., [5, 18, 28, 30, 31]).

Zhang et al. [41] show that dependencies between tests can hinder the applicability and effectiveness of test manipulation techniques. This includes not just TCP and RTS techniques, but also test execution techniques [22, 27], test factoring techniques [33, 39], and test carving techniques [10].

In this work, we focus on test dependencies that may arise following the application of a TCP technique, in which the tests in a test suite are re-ordered for an execution on a subsequent version of a program. When a test is dependent on another test, changing the order in which these tests execute can cause one of them to fail to execute as intended. In such situations, we say that this test “breaks”, and that a “test breakage” has occurred.

Zhang et al. [41] present a formal definition of the test dependency problem. Here, we present that definition more briefly and in a context appropriate to dependencies that affect regression testing. Let $T = \{t_1, t_2, \ldots, t_n\}$ be an original test suite, where each $t_i$ is a test case. Suppose that when tests in $T$ are executed in their original order (the order just listed) all tests execute as intended. Suppose that a TCP technique is applied to $T$, and this changes the execution order of some tests. Suppose this action causes some test $t_v \in T$ to fail to execute as intended, and this failure is due to the fact that $t_v$ depended on the actions performed by some other test $t_w \in T$ that was formerly positioned before $t_v$ and is now positioned after it. In this case we say that $t_v$ is dependent on $t_w$, and we may represent it notionally as $(t_w \rightarrow t_v)$.

The process of manually identifying test dependencies can be challenging. In fact, Zhang et al. [41] have shown that the “Dependent Test Detection Problem” is NP-complete. Given a test suite $T$, this problem involves determining whether a test $t_i \in T$ is dependent in $T$. Thus, previous research has focused on seeking heuristics to provide a cost-effective approximation to detect test dependency.

Previous work reported in DTDectector [11], ElectricTest [7], provides test dependency detection approximation techniques for Java projects. These techniques attempt to identify dependencies in general, by analyzing an application’s code and monitoring read/write operations made to the application’s shared global variables. The goal of these techniques is to identify all test dependencies that may exist in a test suite. PRADET [12] introduced a refinement technique to be applied to the dependency list produced by DTDetector and ElectricTest in order to extract manifest test dependencies that may cause test failures.

In [8], TEDD is presented as a tool for test dependency detection in E2E web application tests. TEDD performs string and natural language processing (NLP) analysis to monitor read-after-write operations and eventually build a test dependency graph (TDG) with all test dependencies. As in PRADET [12], TEDD refines the TDG in

Manuscript received August 18, 2021; revised March 15, 2022. A. Alakeel is a professor of Computer Science Department, University of Tabuk, Tabuk 71491, Saudi Arabia (e-mail: alakeel@ut.edu.sa).
order to only report manifest test dependencies to the developer. All of DTDeectector, ElectricTest, PRADET, and TEDD do not provide any test dependency repair and leaves this tedious and time-consuming process to be performed manually by the developer. In contrast, our work provides an automated and integrated environment for test dependencies detection and repair in web application test suites.

During our investigation of test dependency in web application test suites, we face more changeless other than the read-after-write situations investigated in \cite{7,8,12,41}. For example, during their work with shared global variables in Java projects, \cite{7,12,41} did not have to deal with situations where a shared global variable has been deleted and ceases to exists; they only watch value changes of a shared global variable. In contrast, our work deals with many situations on which a web-page element may deleted, modified or created during the course of executing a test suite. Additionally, in Java projects, application code, test code, and dependency detection tools are encapsulated as a single Java project, while web applications we investigate in this work are written in PHP and a mixture of HTML, JavaScript, CSS, and mySQL. For these reasons, the proposed methods in DTDeectector \cite{41}, ElectricTest \cite{7} and PRADET \cite{12} are not applicable in our context.

Similarly, TEDD \cite{8} only investigates test dependency that may exist as a result of read-after-write operations in a web application’s database, without handling those situations on which a web page element may ceases to exists. In this work, however, we explicitly handle such situations on which a web-page element may be deleted and ceases to exists and also other situations on which a web-page element may be referenced before it has been created. Additionally, TEDD considers that “assert” statements to be the sole source of creating test dependency (\cite{8}, p. 2), and therefore limit its test dependency search on assertion statements only. In our work, WebTestRepair considers all web test statements, including assertions, during its search for test dependencies, and, therefore covers a wider spectrum of web tests, which makes it more practical than TEDD.

In this paper, we present WebTestRepair, the first dynamic and automated test dependency detection and repair integrated environment for web applications. As will be explained in great details in Section \textbf{III}, WebTestRepair focuses on providing an automated solution for a broken test suite. To achieve this goal, WebTestRepair first executes a test suite in order to identify broken tests and then finds those test dependencies that caused these breakages. In order to search for test dependencies that cause tests failures in a test suite, i.e., manifest test dependencies \cite{12}, WebTestRepair performs a combined string and data-flow analysis. As compared to exiting work reported in \cite{7,8,12,41}, that only report detected manifest test dependencies and stop there, a major contribution of WebTestRepair is that it utilizes the detected manifest test dependencies to automatically provide the developer with a repaired test suite that can be executed without test breakages that previously caused by test dependencies.

Note that our approach does not attempt to solve the NP-complete problem that is referred to in \cite{41} as “Dependent Test Detection Problem,” because we are interested in finding only those dependencies that cause observable case breakages in prioritized or re-ordered test suites. Therefore, WebTestRepair does not perform an exhaustive search, nor does it attempt to identify all dependencies. Instead, it concentrates on the detection and repair of those test dependencies that caused some tests to fail. Following this strategy, WebTestRepair produces a quick and feasible repair for a specific set of test case breakages that occur due to test case re-ordering. Thus, it is bounded in terms of execution time.

WebTestRepair is implemented in Java as an integrated automated environment that accepts test suites (original and re-ordered) as input, and automatically detects and repairs test case dependencies in these suites. The current version of WebTestRepair operates on test suites that are created by Selenium-IDE \cite{35} and then exported in the “Java/Unit 4/ WebDriver Backed” format. (Selenium IDE is an open-source tool that is widely used in web application testing, and currently can export test cases in 19 different formats.) The algorithm and its implementation can easily be adapted, however, to handle other formats.

To evaluate the performance of WebTestRepair, we conducted an empirical study on five open-source web applications, in which we applied WebTestRepair to a large number of test suites that have been re-ordered in ways that expose test dependencies. Our results show that WebTestRepair was able to identify and repair test dependencies in 95% of the scenarios in which they occurred, allowing re-ordered test suites to run properly. WebTestRepair was also over 81% effective than a technique that randomly re-orders test cases in an attempt to repair test breakages.

This paper makes the following contributions:

• The first automated execution-based algorithm, to the best of our knowledge, to detect and repair test dependency related failures in web applications’ test suites.

• An automated web application testing framework that includes an implementation of our algorithm in a form that accepts web application test cases generated by Selenium IDE, detects and repairs dependencies among those test cases, and provides support for test case prioritization.

• An empirical study evaluating our algorithm on several non-trivial web applications.

The remainder of this paper is organized as follows. Section \textbf{II} provides background information. Section \textbf{III} presents WebTestRepair and an example of its use. Section \textbf{IV} presents our empirical study and results. In Section \textbf{V} we discuss our results and their implications. Section \textbf{VI} describes related work and Section \textbf{VII} presents conclusions and future work.

\section{Background}

As noted in Section \textbf{II}, this work focuses on tests created using Selenium IDE \cite{35}. Selenium IDE allows developers to record their interactions with a web application.
TABLE I: A Test Suite Specification for the Schoolmate App

<table>
<thead>
<tr>
<th>Test</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>Add a user account of type “Admin”, three accounts of type “Teacher” and one account of type “Student”.</td>
</tr>
<tr>
<td>$t_2$</td>
<td>Fill in personal information for all user accounts.</td>
</tr>
<tr>
<td>$t_3$</td>
<td>Create two terms and in each term create a semester.</td>
</tr>
<tr>
<td>$t_4$</td>
<td>Add a new user account of type “Parent” and update school information.</td>
</tr>
<tr>
<td>$t_5$</td>
<td>Add two new classes.</td>
</tr>
<tr>
<td>$t_6$</td>
<td>Delete a term.</td>
</tr>
</tbody>
</table>

These recordings may then be re-played during testing or regression testing to help expose faults. For example, consider a test suite specified as shown in Table I for the web application “Schoolmate” [34]. Schoolmate is an open-source school management web application that allows users to perform school-related tasks such as creating terms, classes, and semesters, registering in classes, and generating students’ progress reports. It also supports the addition of different types of users including teachers, students, and parents.

Suppose a developer has created a Selenium IDE test suite $T = (t_1, t_2, t_3, t_4, t_5, t_6)$ corresponding to the test specification in Table I. Suppose that when $T$ is executed in this order all tests pass. Assume the developer applies a TCP technique to $T$, and this technique produces a new test suite $T' = (t_4, t_2, t_5, t_6, t_1, t_3)$. When $T'$ is executed in this new order, tests $t_2$ and $t_5$ break (fail to execute as intended) due to dependencies. Test $t_2$ breaks because it is trying to access user accounts that have not yet been created, because the test that creates them, $t_1$, now executes later. Similarly, test $t_5$ breaks because it is trying to add classes; this operation requires a teacher name, which is entered by test $t_2$, and semester information, which is entered by test $t_3$.

Section 3.1 shows how WebTestRepair is able to process test suite $T'$, with broken tests, to eventually produce a working ordering for $T'$ as repaired $T' = (t_4, t_1, t_3, t_2, t_5, t_6)$, such that when repaired $T'$ is executed in this order all tests pass. Note that all the tasks of test dependency detection, repair, and also the application’s execution operations are totally automated by WebTestRepair with no intervention by the developer.

III. TEST DEPENDENCY DETECTION AND REPAIR

Given a prioritized test suite, $T$, in which some tests fail due to test dependencies, WebTestRepair performs a combined string and data-flow analysis to search for manifest test dependencies that cause these failures. Once these manifest test dependencies have been identified, WebTestRepair automatically repair $T$ by producing a workable ordering for it such that when $T$ is executed in this order, all tests in $T$ pass. Furthermore, WebTestRepair performs a validation of its work by automatically executing the repaired test suite on the target web application (Section 3.1-B provides a full description of WebTestRepair, and in Section 3.1-C we present a running example of using WebTestRepair). By providing an automated and integrated solution to the multiple tasks of broken tests identification, the detection of manifest test dependencies, and repair of broken test suites, WebTestRepair relieves the developer of a tedious and labor-intensive repair process and also provides her with an extra assurance on the produced solution. Therefore, WebTestRepair provides more automated services as compared to existing techniques reported in [7], [8], [12], [13], which only produce a test dependency list and leave other tasks to be performed manually by the developer.

In this work, our main goal is to provide a solution for a broken test suite. Therefore, WebTestRepair only identifies test dependencies that are known to cause test breakages (failures) in a re-ordered (e.g., prioritized) test suite. More specifically, given a test $t_i$ which has failed because of test dependency, WebTestRepair follows a greedy strategy on which the search process is paused as soon as the first test dependency is identified. Instead of continuing the search for other test dependencies that may exist in a test suite, WebTestRepair triggers the repair process immediately to investigate if this found dependency is the reason for $t_i$ breakage. In contrast, prior work [7], [8], [12], [21], [24], [13], search for all test dependencies in a test suite and then refine the resultant list to extract manifest dependencies that cause test failures. For example, both PRADET [12] and TEDD [8] first create a test dependency graph (TDG) with all test dependencies found in a test suite and then apply a refinement process on TDG. Because the resultant dependency list may be very large, these techniques apply multiple filtration stages on TDG in order to only maintain “manifest” test dependencies which cause test fails, while excluding others that may not cause test failures [12], [8]. Since WebTestRepair from the outset only searches for manifest test dependencies, it saves the extra time needed for a filtration stage and therefore, it is more efficient than PRADET [12] and TEDD [8].

A. Manifest Test Dependency Detection

In order to search for manifest test dependencies in a broken test suite, WebTestRepair performs a combined string and data-flow analysis as follows. Given a test suite $T$ with a broken (failed) test $t_b$, WebTestRepair identifies the first command in $t_b$ that caused the breakage. With this command in hand, WebTestRepair performs a string analysis to search for clues to locate shared web-page element’s components between $t_b$ and any other test $t_i \in T$. Once these shared web-page elements are identified, WebTestRepair investigates what actions(write/read/delete) have been performed by $t_i$ on those shared elements and based on that, it decides whether a manifest test dependency between $t_i$ and $t_b$ exists or not.

To guide the search process, WebTestRepair identifies a three fine-grained data-flow dependencies as follows. read-after-write (RaW), referenced-after-delete (RaD), and referenced-before-create (RbC). Using this classification, WebTestRepair decides the existence of data-flow dependencies between test $t_i$ and the broken test $t_b$ as follows. A (RaW) data-flow dependency exists between $t_i$ and $t_b$ ($t_i \rightarrow t_b$), if $t_b$ reads the value of a web-page element that has been written by $t_i$. Provided that a web-page element exists previously, there exists a (RaD)
data-flow dependency between $t_1$ and $b_2$ ($t_1 \rightarrow b_2$), if $b_2$ tries to reference (read, write, or delete) a web-page element that has been deleted by $t_1$. A (RbC) data-flow dependency exists between $t_1$ and $b_2$ ($t_1 \rightarrow b_2$), if $b_2$ tries to reference (read, write, or delete) a web-page element which yet to be created by $t_1$.

To explain the difference between RaD and RbC test data-dependencies, consider the Schoolmate web application’s test suite specification introduced previously in [1]. For example, if $t_2$ tries to fill a personal details web-page for a user account that has not been created by $t_1$, then there is a RbC data-flow dependency between $t_2$ and $t_1$ ($t_2 \rightarrow t_1$). On the other hand, if another test, $t_n$, tries to reference (read/write/delete) a term that was created before by $t_3$ but was later deleted by $t_n$, then there exits a RaD data-flow dependency between $t_n$ and $t_3$ ($t_n \rightarrow t_3$).

In this work, WebTestRepair refreshes the application’s database to an empty state before each test suite’s execution. We consider this assumption in order to focus on resolving test dependency on a clean set-up execution’s environment and to avoid state polluting tests situation [14], on which data changes, during different executions, might cause problematic test’s behaviors. This assumption, however, may be relaxed, and an application’s state could be initialized to any appropriate value.

### B. The WebTestRepair Algorithm

Algorithm 1 presents the main routine of the WebTestRepair algorithm, Algorithm 2 presents the algorithm’s primary processing routines, and Algorithm 3 presents supporting routines.

The main procedure WebTestRepair accepts four parameters as input and returns a list as an output. The input parameters are: (1) a reference to a web application $A$, (2) an original test suite $T_o$ in which no tests break, (3) a prioritized version $T_m$ of $T_o$ that contains tests that break due to dependencies, and (4) the maximum number of iterations, $I$, the developer is willing to allocate for WebTestRepair to perform the process of test dependency detection and repair on the input test suites. The output of WebTestRepair is a list, $T_f$, that contains a repaired version (a working ordering) of the input test suite $T_m$. In addition to this output parameter, WebTestRepair produces several output files that contain detailed reports and statistics of its work. Information in these files include: the number of broken tests in each execution, the execution trace for broken test cases, the repaired version of the broken test suite, the execution trace for broken test cases, the repaired version of the broken test suite.

### Algorithm 1 WebTestRepair Algorithm – Main Routine

1: procedure WebTestRepair ($A$, $T_o$, $T_m$, $I$): $T_f$
2: $A$: a web application
3: $T_o$: base test suite with all test cases passing
4: $T_m$: prioritized version of $T_o$ with dependencies
5: $I$: number of iterations
6: $T_r$: repaired version of $T_m$
7: $traceT_r$ ← PreProcess($A$, $T_o$)
8: $traceT_m$ ← PreProcess($A$, $T_m$)
9: $finished ← false$
10: $traceT ← traceT_m$
11: while (not $finished$ & $I > 1$) do
12: $T_e ← DepRepair(traceT_m, TraceT_m)$
13: if $T_e$ is empty then
14: exit while loop
15: else
16: if (all test cases in $T_e$ pass) then
17: $finished ← true$
18: report success
19: return $T_r$
20: else
21: $I ← I − 1$
22: $traceT ← Execute(A, T_r)$
23: end if
24: end if
25: end while
26: if (not $finished$) then
27: report failure to repair failed suite
28: end if
29: end procedure

### Algorithm 2 WebTestRepair Algorithm – Primary Sub-routines

1: procedure PreProcess ($A$, $T$): $traceT$
2: $A$: a web application
3: $T$: a test suite for $A$
4: $traceT$: trace for $T$ on $A$
5: $preT ← Prepare(T, preT)\$ traceT ← Instrument(preT)\$
7: $traceT ← Execute(A, T)$
8: return $traceT$
9: end procedure
10: procedure DepRepair ($traceT_m$, $TraceT_m$): $T_r$
12: $traceT_r$: execution trace for original test suite
13: $traceT_m$: execution trace for broken test suite
14: $T_r$: a repaired version of the broken test suite
15: $bTests ← AllBroken(traceT_r)$
16: $allBdeplists ← empty$
17: for (each test case $b_i$ in $bTests$) do
18: $oTrace ← GetTrace($b_i$, traceT_m)$
19: $pS ← ProblemS($b_i$, oTrace)$
20: $index ← Location($oTrace$)
21: $bDeplist ← empty$
22: $bDeplist ← AddFirst($b_i$)
23: for (each test case $t_i$ in $traceT_m$ backwards from index) do
24: if (Referenced($pS$, $t_i$)) then
25: $bDeplist ← AddFirst($t_i$)
26: end if
27: end for
28: $allBdeplists ← Add($bDeplist$)
29: end for
30: $T_r ← RepairedList(allBdeplists, traceT_m)$
31: return $T_r$
32: end procedure
33: procedure RepairedList ($allBdeplists$, $traceT_m$): $T_f$
35: $allBdeplists$: list of dependency lists for broken test cases
36: $traceT_m$: execution trace for broken test suite
37: $T_f$: a test suite in which all test cases pass on $A$
38: $T_f ← empty$
39: for (each test case $t_i$ in $exeT_m$ backwards from tail) do
40: if (Broken($t_i$) then
41: $bDeplist ← dependency list of $t_i$ from $allBdeplists$
42: for (each test case $t_k$ in $bDeplist$ backwards from tail) do
43: $t_k ← AddFirst($t_k$)
44: end if
45: end for
46: else
47: $t_k ← AddFirst($t_k$)
48: end if
49: $T_f ← AddFirst($t_k$)
50: end if
51: end if
52: end for
53: return $T_f$
54: end procedure

Volume 49, Issue 2: June 2022
Algorithm 3 WebTestRepair Algorithm – Auxiliary Methods

1: procedure PREPARE (T): preT 
2: Input: T, a test suite in Selenium IDE format in which test cases are saved as text files 
3: Output: preT, a prepared test suite for which each test is wrapped as a Java Unit4 class 
4: end procedure 
5: 
6: procedure INSTRUMENT (T): T' 
7: Input: T, a test suite for A 
8: Output: T', an instrumented version of T 
9: end procedure 
10: 
11: procedure EXECUTE (A, T): traceT 
12: Input: A, a web application 
13: Output: traceT, an execution trace for T on A along with execution results for all test cases in T 
14: end procedure 
15: 
16: procedure ALLBROKEN (traceTm): bTtests 
17: Input: traceTm, the execution trace for the broken test suite 
18: Output: bTtests, a list of broken test cases in Tm 
19: end procedure 
20: 
21: procedure GETTRACE (t, oTrace): oTrace 
22: Input: t, a broken test case 
23: Output: oTrace, the execution trace for the broken test suite 
24: end procedure 
25: 
26: procedure PROBLEMS (t, oTrace): pS 
27: Input: t, a broken test case 
28: Output: pS, the statement in t that caused t to break 
29: end procedure 
30: 
31: procedure REFERENCED (pS, t): result 
32: Input: pS, the statement in test case t that caused t to break 
33: Output: result, true if pS was read, written, or modified by t, false otherwise 
34: end procedure 
35: 
36: procedure BROKEN (t): result 
37: Input: t, a test case 
38: Output: result, true if t is broken, false otherwise 
39: end procedure 
40: 
41: 
42: 
43: 
44: 

The PreProcess procedure (Algorithm 2) is responsible for setting up the processing environment for WebTestRepair. This procedure accepts two parameters as an input and returns a text file as an output. The first parameter is the web application A and the second one is a test suite T to be executed on A. The output of PreProcess is an execution trace of T. Additionally, PreProcess produces other internal output files which will be used by WebTestRepair during its operations.

When PreProcess is called with test suites T₀ and Tₘ, and because tests in these suites are originally generated by Selenium IDE as Java/Unit4/WebDriver Backed format, as stated in Section refsec:introduction, PreProcess first transforms these files into a Java/Unit4 code format suitable to run in the Java Eclipse execution environment. For example, Table III shows a Selenium generated Java/Unit4/WebDriverBacked code for test (t₁) created from the test specification previously shown in Table I. For this purpose, PreProcess (in Line 5 of Algorithm 2) invokes the Prepare procedure (see Algorithm 3). During its work, the Prepare procedure scans the input file to remove white spaces and unwanted statements generated by Selenium IDE, retaining only the codes testaments of this test. Secondly, (in Line 6 of Algorithm 2) PreProcess makes a call to Instrument procedure (Algorithm 3) that will insert instrumentation’s statements necessary to record the test’s execution trace and also inserts run-time library’s imports and creates the Java/Unit4 Before and After methods.

Once all Java/Unit4 classes have been prepared for all the input test suites, T₀ and Tₘ, the PreProcess invokes the Execute routine (Algorithm 3) to execute all of these Java/Unit4 classes and returns their execution trace to the calling procedure, WebTestRepair. The information reported in the execution trace files includes the execution’s ordering of tests within the current test suites, the execution’s result of each test (“pass” or “fail”), and also a list of executed commands for each test.

The Instrument procedure performs two tasks to help the calling procedure, PreProcess, in building a Java/Unit4 test class file for each input test. The first one is to instrument each code’s statement with the necessary code to capture its execution trace, and the second task is to parse each test’s command to construct an internal representation that WebTestRepair utilizes during its search for manifest test dependencies on later stages of the algorithm.

Give a Selenium test T with a set of commands C= {c₁, c₂, . . . , cₙ}, for each command cₖ ∈ C, we create a data-structure cmdRep(cₖ) = <type, locator, element identifier, argument, argument value, hooksUp, hooksDown>. This internal representation captures the five main components of each command in addition to two other components (hooksUp, hooksDown) that provide additional information about surrounding commands of cₖ within the same test T. Figure IV displays an internal representation for the Selenium command in Line 12 of Table III. In this work, we only consider the following five Selenium command types: (open, type, click, select, and assert), while we ignore others such as

called PreProcess. The purpose of these two calls is to preprocess the original test suite T₀ and its prioritized version Tₘ, respectively. PreProcess ultimately returns an execution trace for T₀ or Tₘ. Next, in the while loop of Lines 11–25, WebTestRepair calls DepRepair (Line 12), which attempts to detect test dependencies that caused some test in Tₘ to break. This while loop is performed until either a repaired test suite is found or the maximum number of iterations is reached. WebTestRepair terminates as soon as a repaired test suite is found; thus it is not required to complete all iterations.

In some difficult situations, however, WebTestRepair may not succeeds during its mission to identify all test dependencies causing tests failures in the currently processed suite as indicated by Lines 13–14. Note that WebTestRepair builds its solution incrementally and may finish its work before reaching the maximum number of iterations. However, depending on the difficulty of the processed test suites, the number of iterations may influence WebTestRepair effectiveness and efficiency, i.e., allowing WebTestRepair to run for more iterations may increase the chance of producing a successful solution. For example, in our experiment presented in Section IV, we allowed WebTestRepair to run for a maximum of five iterations.
TABLE II: Java/Unit 4/WebDriver/Backed code exported from Selenium-IDE for Test Case t₁

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>selenium.open(&quot;/schoolmate/Test&quot;)</code>;</td>
</tr>
<tr>
<td>2</td>
<td><code>selenium.type(&quot;name=username&quot;, &quot;test&quot;);</code></td>
</tr>
<tr>
<td>3</td>
<td><code>selenium.type(&quot;name=password&quot;, &quot;test&quot;);</code></td>
</tr>
<tr>
<td>4</td>
<td><code>selenium.click(&quot;css=input[type=submit]&quot;);</code></td>
</tr>
<tr>
<td>5</td>
<td><code>selenium.waitForPageToLoad(&quot;60000&quot;);</code></td>
</tr>
<tr>
<td>6</td>
<td><code>selenium.click(&quot;link=Users&quot;);</code></td>
</tr>
<tr>
<td>7</td>
<td><code>selenium.waitForPageToLoad(&quot;60000&quot;);</code></td>
</tr>
<tr>
<td>8</td>
<td><code>selenium.click(&quot;xpath=[/input[@value=Add]]&quot;);</code></td>
</tr>
<tr>
<td>9</td>
<td><code>selenium.waitForPageToLoad(&quot;60000&quot;);</code></td>
</tr>
<tr>
<td>10</td>
<td><code>selenium.type(&quot;name=username&quot;, &quot;aley&quot;);</code></td>
</tr>
<tr>
<td>11</td>
<td><code>selenium.type(&quot;name=password&quot;, &quot;aley&quot;);</code></td>
</tr>
<tr>
<td>12</td>
<td><code>selenium.type(&quot;name=password2&quot;, &quot;aley&quot;);</code></td>
</tr>
<tr>
<td>13</td>
<td><code>selenium.click(&quot;css=input[type=button]&quot;);</code></td>
</tr>
<tr>
<td>14</td>
<td><code>selenium.waitForPageToLoad(&quot;60000&quot;);</code></td>
</tr>
<tr>
<td>15</td>
<td><code>selenium.click(&quot;xpath=[/input[@value=Add]]&quot;);</code></td>
</tr>
<tr>
<td>16</td>
<td><code>selenium.waitForPageToLoad(&quot;60000&quot;);</code></td>
</tr>
<tr>
<td>17</td>
<td><code>selenium.type(&quot;name=username&quot;, &quot;moody&quot;);</code></td>
</tr>
<tr>
<td>18</td>
<td><code>selenium.type(&quot;name=password&quot;, &quot;mary&quot;);</code></td>
</tr>
<tr>
<td>19</td>
<td><code>selenium.type(&quot;name=password2&quot;, &quot;mary&quot;);</code></td>
</tr>
<tr>
<td>20</td>
<td><code>selenium.select(&quot;name=type&quot;, &quot;label=Student&quot;);</code></td>
</tr>
<tr>
<td>21</td>
<td><code>selenium.click(&quot;css=input[type=button]&quot;);</code></td>
</tr>
<tr>
<td>22</td>
<td><code>selenium.waitForPageToLoad(&quot;60000&quot;);</code></td>
</tr>
<tr>
<td>23</td>
<td><code>selenium.click(&quot;xpath=[/input[@value=Add]]&quot;);</code></td>
</tr>
<tr>
<td>24</td>
<td><code>selenium.waitForPageToLoad(&quot;60000&quot;);</code></td>
</tr>
<tr>
<td>25</td>
<td><code>selenium.type(&quot;name=username&quot;, &quot;moody&quot;);</code></td>
</tr>
<tr>
<td>26</td>
<td><code>selenium.type(&quot;name=password&quot;, &quot;moody&quot;);</code></td>
</tr>
<tr>
<td>27</td>
<td><code>selenium.type(&quot;name=password2&quot;, &quot;moody&quot;);</code></td>
</tr>
<tr>
<td>28</td>
<td><code>selenium.select(&quot;name=type&quot;, &quot;label=Teacher&quot;);</code></td>
</tr>
<tr>
<td>29</td>
<td><code>selenium.click(&quot;css=input[type=button]&quot;);</code></td>
</tr>
<tr>
<td>30</td>
<td><code>selenium.waitForPageToLoad(&quot;60000&quot;);</code></td>
</tr>
<tr>
<td>31</td>
<td><code>selenium.click(&quot;xpath=[/input[@value=Add]]&quot;);</code></td>
</tr>
<tr>
<td>32</td>
<td><code>selenium.waitForPageToLoad(&quot;60000&quot;);</code></td>
</tr>
<tr>
<td>33</td>
<td><code>selenium.type(&quot;name=username&quot;, &quot;u1&quot;);</code></td>
</tr>
<tr>
<td>34</td>
<td><code>selenium.type(&quot;name=password&quot;, &quot;u1&quot;);</code></td>
</tr>
<tr>
<td>35</td>
<td><code>selenium.type(&quot;name=password2&quot;, &quot;u1&quot;);</code></td>
</tr>
<tr>
<td>36</td>
<td><code>selenium.select(&quot;name=type&quot;, &quot;label=Teacher&quot;);</code></td>
</tr>
<tr>
<td>37</td>
<td><code>selenium.click(&quot;css=input[type=button]&quot;);</code></td>
</tr>
<tr>
<td>38</td>
<td><code>selenium.waitForPageToLoad(&quot;60000&quot;);</code></td>
</tr>
<tr>
<td>39</td>
<td><code>selenium.click(&quot;xpath=[/input[@value=Add]]&quot;);</code></td>
</tr>
<tr>
<td>40</td>
<td><code>selenium.waitForPageToLoad(&quot;60000&quot;);</code></td>
</tr>
<tr>
<td>41</td>
<td><code>selenium.type(&quot;name=username&quot;, &quot;u2&quot;);</code></td>
</tr>
<tr>
<td>42</td>
<td><code>selenium.type(&quot;name=password&quot;, &quot;u2&quot;);</code></td>
</tr>
<tr>
<td>43</td>
<td><code>selenium.type(&quot;name=password2&quot;, &quot;u2&quot;);</code></td>
</tr>
<tr>
<td>44</td>
<td><code>selenium.select(&quot;name=type&quot;, &quot;label=Teacher&quot;);</code></td>
</tr>
<tr>
<td>45</td>
<td><code>selenium.click(&quot;css=input[type=button]&quot;);</code></td>
</tr>
<tr>
<td>46</td>
<td><code>selenium.click(&quot;css=input[type=submit]&quot;);</code></td>
</tr>
<tr>
<td>47</td>
<td><code>selenium.waitForPageToLoad(&quot;60000&quot;);</code></td>
</tr>
</tbody>
</table>

(While `waitForPageToLoad` and `soft` do not provide any useful information to guide the search for manifest test dependencies. A web page element locator may take the values (xpath, css, id, name, and link). The upper hooks, `hooksUp`, is a list of preceding "click" commands, sorted backward up to the first command in the test, while `hooksDown` is a list of succeeding "click" commands, sorted forward down until the last command in the test. The hook lists are utilized by the algorithm to widen its search for manifest test dependencies to the surrounding commands, when there is not enough information provided by the command's main components to guide the search. Note that the values of `cmdRep(eₙ)` varies from one command to another depending on its type. Also, all or only some of these components may be utilized by the algorithm during the search of test dependencies, depending on the difficulty of the problem statement which caused the test breakage.

After the pre-processing stage is done, `WebTestRepair`, in (Line 12), calls the `DepRepair` procedure (Algorithm 2) with `traceT₀` and `traceTₘ`, the execution traces for `T₀` and `Tₘ`, respectively. `DepRepair` performs a combined string and data-flow analysis on these execution traces to identify test dependencies that caused test breakages in `Tₘ`. After identifying test dependencies, `DepRepair` uses `T₀` as an oracle and starts a repair process for `Tₘ` trying to produce a working execution ordering for `Tₘ` such that when executed in this order all tests pass. Note that `traceTₘ` contains all pass execution trace, while `traceTₘ` includes broken tests execution's trace.

Lines 11–32 of Algorithm 2 describe `DepRepair`'s actions more precisely. When `DepRepair` is called with execution traces, `traceT₀` and `traceTₘ`, `DepRepair` first calls (Line 15) an auxiliary routine AllBroken to obtain `bTests`, which is a list of broken (failed) tests found in the execution trace `traceTₘ`. Next, in the for loop of Lines 17–29, `DepRepair` considers each broken test in `bTests`. For each broken test `tᵢ`, `DepRepair` calls `GetTrace` (Algorithm 3) to obtain a pass execution trace for `tᵢ` from `traceT₀`, which will be utilized later to locate the breakage's location in `tᵢ`.

In order to locate the breakage's location in `tᵢ`, `DepRepair` (Line 19) calls the `ProblemS` procedure (Algorithm 3), which identifies the first code statement `pS` that caused the breakage in test `tᵢ`. We refer to this `pS` as the "problem statement". To locate `pS` in `tᵢ`, the `ProblemS` procedure, performs string-analysis on `tᵢ`'s execution trace (pass trace) obtained form `traceT₀`, and its other trace (fail trace) obtained from `traceTₘ`. Next, in line 20 of Algorithm 2 `DepRepair` obtains an index that lists the order of `tᵢ` when executed in `traceT₀`. At this stage, `DepRepair` triggers the test dependency detection process in (Line 22) creating a dependency linked list data structure, `bDepList`, for `tᵢ` and places test `tᵢ` as the head element of this linked list.

To identify manifest test dependencies causing the breakage of test `tᵢ`, `DepRepair` considers each test `tᵢ` that was executed before `tᵢ` from the original test suite execution trace, `traceT₀`, as shown by the code in the for loop of Lines 23–27. Specifically, for each `tᵢ`, `DepRepair` calls the `Referenced` procedure (Algorithm 3) which performs a combined string and data-flow analysis to search for manifest test dependency between each test `tᵢ` and `tᵢ`.

The referenced procedure begins the process of manifest test dependency detection by performing a string-analysis searching each test `tᵢ` for clues that may lead to locating any links between the shared web element that caused `tᵢ` breakage and any of these tests. At first, the string-analysis search is based on `pS` main components, i.e., the command's type, the locator, element's identifier, the argument, and the argument value. When this information is not enough to guide this search, the algorithm widen its search through the upper hooks list to inspect surrounding commands. Again, if this is not successful, then search continues using the lower hooks list. The hook lists provide the algorithm with more information because each command in these list performs a "click" operation within the same test. Therefore, the purpose of searching these lists is to identify other command(s) within the broken test that might be related to the problem statement `pS`, and therefore providing the algorithm with more clues to fully understand the operations performed by `pS`, when the search using...
its main components fails. We employ this heuristic in order to handle those challenging situations on which the web application developer uses different locators to identify the same web element and also does not follow a meaningful naming mechanism for elements identifiers.

If the string analysis search in not successful to identify any shared web elements between \( t_b \) and any test \( t_i \) that was executed before it in the original test suite, the result is reported and the algorithm continues to process the remaining broken tests, if any. On the other hand, if the string analysis search is successful then the result of this stage reports a list of tests that have been identified to have some links to that web element causing \( t_b \) to break. At point, Referenced begins a data-flow analysis process to search for manifest test dependencies between each test \( t_i \) in this list and \( t_b \).

The data-flow dependency search is performed in order to identify the type of actions performed by each test \( t_i \) on the web page element that has been identified to cause \( t_b \) breakage. During this search, the algorithm inspects each command searching for the following actions: create, select (read), write, delete, or to any other textual clues that might lead to the identification of any of them. For example, words like "add," "new," and "insert" are clues for a "create" action, words like "update" or "edit" are clues for a "write" action, words like "remove", "drop" are clues for a "delete" action, while a word like "select" is a clue for a "read" action. Secondly, based on the results of this search, Referenced identify the type of the data-flow dependency (i.e., RaW, RaD, or RaB) between \( t_i \) and \( t_b \), using the data-flow dependency classification defined in Section II-A. This extra step in necessary because a shared web page element between \( t_i \) and \( t_b \) may create a data dependency between these tests, but does not necessarily implies the existence of a manifest test dependency [12]. Therefore, WebTestRepair main goal is to identify manifest test dependencies while ignoring data dependencies which don't cause test breakages. Once a data-flow dependency is found between \( t_i \) and the broken test \( t_b \), then this implies that a manifest test dependency \( (t_i \rightarrow t_b) \) has been identified.

To illustrate the use of the hook lists, consider the code in Figure 2 which shows one of the php-Addressbook application test cases used in our experiment. In this example, the select command in Line 4 has been identified, by the algorithm, to be the cause for this test failure. When the algorithm inspects this command internal components looking for a textual clue to determine the web element manipulated by this command, it will get stuck because there is no textual clues in this command to guide the string analysis search. Therefore, the algorithm inspects the internal representation for this failed command and utilizes the information provided in the hook lists in order to search surrounding commands for any clues that may lead to the indemnification of a textual representation for this web element. Given the failed command in Line 4 and as explained earlier, the upper hooks list will includes the code in Line 2 and the lower hooks list includes the code in Line 9. Given this information, the algorithm first inspects the upper hook list and eventually identifies the string "groups" in Line 2 as a textual clue for the web element that is been manipulated by the failed command in Line 4. At this point, the algorithm uses this clue (i.e., "groups") during its subsequent search for test dependencies between this broken test and other tests in the test suite under processing. Further, the algorithm inspects the lower hooks list and utilizes the string "update," in Line 9, as a clue that leads the flow-analysis search to determine the action that has been performed, by the failed command in Line 2, on the selected element (i.e., "groups"). For this example, note that the algorithm has succeeded to identify a textual clue for the manipulated web element because the failed statement in Line 4 actually involves the selection of a "groups" list.

After the Referenced procedure finishes its work, DepRepair (Line 28) is able to construct a set of test dependency lists, allBdeplists, for all broken tests. At this point (in (Line 30), DepRepair calls another procedure called RepairedList (Algorithm 2) to trigger the test dependency repair process. Eventually, when the RepairedList procedure work is completed, it returns a repaired test dependency list to the main routine WebTestRepair as indicated in (Line 31) of Algorithm 2.

The RepairedList is called by DepRepair with two input parameters: allBdeplists and trace\(T_m\), a list of lists that contains the dependency lists, bDeplists, for all broken tests case and the execution traces for these broken tests, respectively. Lines 34–54 of Algorithm 2 describe RepairedList’s actions more precisely. In Line 38, RepairedList initializes an empty list, \( T_f \), which will hold the repaired test suite. Next (Lines 39–52), RepairedList considers each test case in \( traceT_m \) from the last test case to the first as follows. For each test case \( t_i \), RepairedList calls a supporting routine, Broken, to determine whether \( t_i \) was broken or not. If Broken(\( t_i \)) is true, RepairedList extracts bDeplist, the dependency list for \( t_i \), from allBdeplists, and begins a new loop in Line 42. RepairedList next (Lines 42–46) considers each test case \( t_d \) in bDeplist, beginning with the last test case and proceeding backward to the first. In Lines 43–45, RepairedList adds \( t_d \) to the head of \( T_f \) if it is not has been added previously.

If Broken(\( t_i \)) is false, RepairedList performs additional checking in Lines 48–50 before it decides whether to add \( t_i \) to \( T_f \) or not. Because a particular test case could appear in multiple dependency lists built by the algorithm, RepairedList performs a check in Line 48 to prevent a specific test case from being added in duplicate into \( T_f \). Finally, in Line 53, RepairedList returns the repaired test suite, \( T_f \), to DepRepair, which returns it to WebTestRepair.

C. A Running Example of WebTestRepair

Consider the two test suites that have been created for the Schoolmate [34] web application, previously introduced in Section I. These test suites are: \( T_2 = (t_1, t_2, t_3, t_4, t_5, t_6) \) and its prioritized version \( T_m = (t_4, t_2, t_3, t_5, t_6, t_1) \). Recall, from Section I that these tests are created based on the specification of Table I and that both \( t_5 \) and \( t_5 \) failed during the execution of \( T_m \) due to test dependency.

Volume 49, Issue 2: June 2022
To process these two test suites, WebTestRepair performs the following fully automated steps:

1) Using a local host as a web-server environment, WebTestRepair first executes $T_o$ on the Schoolmate application in order to confirm that all tests pass and to capture $T_o$ execution’s trace during this process.

2) Execute the prioritized test suite, $T_m$, on Schoolmate in order to detect test breakages and also to capture $T_m$ execution’s trace. In this example, $t_2$ and $t_5$ are going to be identified as failed tests.

3) If step (2) returns test breakages (which is the case in this example), then WebTestRepair performs test dependency detection and repair to eventually produce a working ordering for tests in $T_m$, such that when executed in this new order, all tests pass. For this example, the new working ordering produced by WebTestRepair for, $T_m$, is $(t_4, t_1, t_3, t_2, t_5, t_6)$.

4) Execute the new ordering produced in step (3) to confirm that a solution has been provided.

To perform the above mentioned steps, WebTestRepair starts by executing the main routine WebTestRepair($A, T_o, T_m, I$) with its expected four input parameters as stated in Algorithm 1. For our example, the values for these parameters are: $A =$ Schoolmate, $T_o = (t_1, t_2, t_3, t_4, t_5, t_6)$, $T_m = (t_1, t_2, t_3, t_4, t_5, t_6, t_1)$, and $I = 5$. Notice here that we have allocated a maximum of 5 rounds for WebTestRepair to work on our example.

As explained earlier in Section III-B when called with these parameters, the WebTestRepair routine, calls the PreProcess procedure in order create execution trace for $T_o$ and $T_m$. When finished, PreProcess returns the required traces for $T_o$ and $T_m$ as $traceT_o$ and $traceT_m$, respectively. WebTestRepair now begins the first iteration of the while loop’s execution and calls DepRepair with $traceT_o$ and $traceT_m$; this begins the process of test case dependency detection. Note that WebTestRepair refreshes the Schoolmate database to an empty state before each test suite’s execution.

When called, DepRepair extracts a list of broken tests, $bTests$, from $traceT_m$. For our example, $bTests = (t_2, t_5)$. Next, DepRepair constructs dependency lists for $t_2$ and $t_5$ as follows. First, DepRepair considers the execution trace for $t_2$. Because $t_2$ passed when executed in the original test suite, all of its commands (Lines 1-30), shown in Table II are executed. When $t_2$ is executed in the prioritized suite, $t_2$ breaks in Line 12, and therefore its code in Lines 12-30 will not be included in its prioritized execution’s trace. In this example, excluded commands are italicized in Table III. By comparing a passing trace with a trace related to a breakage, DepRepair determines the statement that caused $t_2$ to break – the ProblemS routine identifies the code in Line 12 of Table III $pS=$ (selenium.select("name=username", "label=u2")), as the problem statement. An internal representation for this command (i.e., $pS$) is shown in Figure 1.

With this problem statement (i.e. Line 12) in hand, DepRepair starts its test dependency search to pinpoint what caused $t_2$’s breakage. During this process, DepRepair needs to know $t_2$ order when executed within the all-pass suite, $T_o$, as described in detail in described in Section III-B. For our example this value is 2, because $t_2$ was the second to execute in $T_o$. DepRepair also needs to know those tests that were executed before $t_2$ in suite $T_m$. For our example, DepRepair finds out that $t_1$ was the only test executed before $t_2$. At this point, using the internal representation of the problem statement ($pS$) in $t_2$, DepRepair performs a combined string and data-flow analysis on $t_1$ to search for manifest test dependency between $t_1$ and $t_2$, as described previously in Section III-B.

The string analysis search finds out that the same web page element "username" that is manipulated by the broken command $pS$ of $t_2$ is also manipulated by the command in Line 41 of test $t_1$ as shown in Table III. Given this information, the algorithm begins a data-flow analysis search to inspect Line 41 command of test $t_1$ in order to find out the type of action this command performs on the shared element, i.e., "username". This inspection reveals that the command in Line 12 of test $t_1$ performs a write operation on the shard web element "username" and sets its value to "u2". At this point and using the classification described previously in Section III-A DepRepair concludes that there is a referenced-before-create (RbC) data-flow dependency between $t_1$ and $t_2$, ($t_1 \rightarrow t_2$). This (RbC) data-flow dependency exists because $t_2$ tries to reference a user account with identifier "username" and value "u2" before it has been created by $t_1$, as shown in Lines 36-45 of Table II.

At this point and after DepRepair confirms the existence of a manifest test dependency between $t_1$ and $t_2$ ($t_1 \rightarrow t_2$), it creates a dependency list for the broken test $t_2$ and inserts as follows. $bdDepList = <t_1 \rightarrow t_2>$. Although there was only one test dependency for $t_2$ in this example, it is possible that multiple dependence may exist. For this reason, DepRepair continues its search for other dependencies that might exist between $t_2$ and other tests following the strategy explained in in Section III-B.

Similarly, DepRepair confirms the existence of an (Rbc) data-flow dependency $t_2$ and $t_5$, $t_2 \rightarrow t_5$, because $t_5$ attempts to create a class for which a teacher’s account with a teacher named “Sara Cameron” has to be created first by $t_2$. At this point, DepRepair constructs a dependency list for $t_5$ as $bdDepList = <t_2 \rightarrow t_5>$. Eventually, as described in Section III-B DepRepair adds this newly generated dependency list to allDepLists, which becomes allBDepLists= $<t_1 \rightarrow t_2, t_2 \rightarrow t_5>$. At this point, DepRepair invokes RepairedList to perform another repair on the test suite produced by the first iteration of the algorithm.

At this point, the RepairedList procedure is called by DepRepair with allBDepLists= $<t_1 \rightarrow t_2, t_2 \rightarrow t_5>$ and $traceT_m$. Beginning with an empty $T_f$, RepairedList considers each test case in the prioritized test suite in backward order beginning from the last one. Thus, it first considers $t_1$, because $t_1$ is the last test case in the prioritized test suite $T_m$. Now, RepairedList determines whether $t_1$ is a broken test case or not. If not (which is the case for $t_1$), $t_1$ is added to $T_f$ only if it is absent from one of the dependency lists for all broken test cases.
TABLE III: Executed Code for Failed Test Case \( t_2 \)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>selenium.open(&quot;/schoolmateTest/index.php&quot;);</td>
</tr>
<tr>
<td>2</td>
<td>selenium.type(&quot;name=username&quot;, &quot;test&quot;);</td>
</tr>
<tr>
<td>3</td>
<td>selenium.type(&quot;name=password&quot;, &quot;test&quot;);</td>
</tr>
<tr>
<td>4</td>
<td>selenium.click(&quot;css=input[type=submit]&quot;);</td>
</tr>
<tr>
<td>5</td>
<td>selenium.waitForPageToLoad(&quot;60000&quot;);</td>
</tr>
<tr>
<td>6</td>
<td>selenium.click(&quot;link=Teachers&quot;);</td>
</tr>
<tr>
<td>7</td>
<td>selenium.waitForPageToLoad(&quot;60000&quot;);</td>
</tr>
<tr>
<td>8</td>
<td>selenium.click(&quot;xpath=/(//input[@value='Add'])[2]&quot;);</td>
</tr>
<tr>
<td>9</td>
<td>selenium.waitForPageToLoad(&quot;60000&quot;);</td>
</tr>
<tr>
<td>10</td>
<td>selenium.type(&quot;name=name&quot;, &quot;Lisa&quot;);</td>
</tr>
<tr>
<td>11</td>
<td>selenium.type(&quot;name=Inn&quot;, &quot;Harry&quot;);</td>
</tr>
<tr>
<td>12</td>
<td>selenium.select(&quot;name=username&quot;, &quot;label=u2&quot;);</td>
</tr>
<tr>
<td>13</td>
<td>selenium.click(&quot;css=input[type=button]&quot;);</td>
</tr>
<tr>
<td>14</td>
<td>selenium.waitForPageToLoad(&quot;60000&quot;);</td>
</tr>
<tr>
<td>15</td>
<td>selenium.click(&quot;xpath=/(//input[@value='Add'])[2]&quot;);</td>
</tr>
<tr>
<td>16</td>
<td>selenium.waitForPageToLoad(&quot;60000&quot;);</td>
</tr>
<tr>
<td>17</td>
<td>selenium.type(&quot;name=faame&quot;, &quot;Sara&quot;);</td>
</tr>
<tr>
<td>18</td>
<td>selenium.type(&quot;name=inn&quot;, &quot;Cameron&quot;);</td>
</tr>
<tr>
<td>19</td>
<td>selenium.click(&quot;css=input[type=button]&quot;);</td>
</tr>
<tr>
<td>20</td>
<td>selenium.waitForPageToLoad(&quot;60000&quot;);</td>
</tr>
<tr>
<td>21</td>
<td>selenium.click(&quot;xpath=/(//input[@value='Add'])[2]&quot;);</td>
</tr>
<tr>
<td>22</td>
<td>selenium.type(&quot;name=faame&quot;, &quot;Moody&quot;);</td>
</tr>
<tr>
<td>23</td>
<td>selenium.type(&quot;name=inn&quot;, &quot;Aley&quot;);</td>
</tr>
<tr>
<td>24</td>
<td>selenium.select(&quot;name=username&quot;, &quot;label=moody&quot;);</td>
</tr>
<tr>
<td>25</td>
<td>selenium.waitForPageToLoad(&quot;60000&quot;);</td>
</tr>
<tr>
<td>26</td>
<td>selenium.click(&quot;link=Log Out&quot;);</td>
</tr>
<tr>
<td>27</td>
<td>selenium.waitForPageToLoad(&quot;60000&quot;);</td>
</tr>
</tbody>
</table>

Because \( t_1 \) is in the dependency list of \( t_2 \), it is not added to \( T_f \).

Next, \( t_5 \) is considered because it was not broken, it is added to the head of the fixed list, which becomes \( T_f = <t_6> \). When \( t_5 \) is considered, because it was broken, RepairedList obtains the dependency list \( bDeplist \) for \( t_5 \) and examines it to determine which test cases should be added to the fixed list, as follows. If a test case does not already exist in \( T_f \) then it will be added to the head of \( T_f \) in the same order it was created in the dependency list for \( t_5 \). Recall that the dependency list for \( t_5 \) was previously found to be \( bDeplist = <t_2 \rightarrow t_5> \). Therefore both \( t_2 \) and \( t_5 \) are added to the head of \( T_f \), which now becomes \( T_f = <t_2, t_5, t_6> \). Next, \( t_3 \) is considered and because it was not broken, it is added to the head of the fixed list, which becomes \( T_f = <t_3, t_2, t_5, t_6> \). When \( t_2 \) is considered, because it is broken, RepairedList retrieves the dependency list for \( t_2 \), \( bDeplist \), and examines it to determine which test cases should be added to the fixed list. Recall that the \( bDeplist \) for \( t_2 \) was found previously to be \( bDeplist = <t_1 \rightarrow t_2> \). Thus, \( t_1 \) is added to the head of \( T_f \) because \( t_2 \) already exists in that list, so now \( T_f = <t_1, t_3, t_2, t_5, t_6> \).

Finally, \( t_4 \) is considered, and is added to the head of \( T_f \) because it is not broken and has not been added previously. The final repaired test suite is \( T_r = <t_4, t_1, t_3, t_2, t_5, t_6> \). This test suite is returned to DepRepair, which returns it to WebTestRepair as \( T_r \).

At this point WebTestRepair performs a confirmation run of \( T_r \) and examines the result of this execution. If no test cases break, WebTestRepair reports the results, the success of the repair process, and quits regardless of the number of iterations remaining. Otherwise, WebTestRepair decrements the number of iterations by one and proceeds with a second iteration in which \( T_m = T_r \). During this iteration, the partially repaired test suite, \( T_r \), will be the target for test case dependency detection and repair by WebTestRepair, while the original test suite continues to be \( T_o \). For our example, we will have \( T_m = \{t_1, t_2, t_3, t_5, t_6\} \) and \( T_o = \{t_1, t_2, t_3, t_4, t_5, t_6\} \). The final contents \( T_m \) is copied into \( T_r \) and returned to the calling procedure, WebTestRepair.

Note that although we allocated a maximum of five iterations in this example, WebTestRepair was able to produce a repaired test suite \( T_r \) that runs without any breakages, using only one iteration, therefore the algorithm reports its result and quits.

D. Implementation

Our algorithm is implemented as a tool called WebTestRepair. WebTestRepair is implemented in Java as a standalone project using JavaSE-1.7 and the Eclipse development environment. In its current state, the tool accepts Selenium test cases in the Java/JUnit4/WebDriverBacked format, but it can easily be extended to handle other test case formats such as JUnit or WebDriver. In our work, each test case is represented by a one-method class. Thus, we can use a test class or test method without losing the intended general meaning of a given test case. WebTestRepair uses its own parser to instrument test cases. During the instrumentation process, an internal representation is created and logged for each Java/JUnit4/WebDriverBacked statement in each test case. We use MAMP [29] to run our subject web applications locally on a local server.

E. Computational Complexity

As noted earlier, WebTestRepair does not attempt to identify all test case dependencies within a given test suite \( T \), and therefore, is computationally bounded. Inspecting the code of WebTestRepair’s main routine presented in Algorithm 1, it can be deduced that the worst case computational complexity of WebTestRepairW is \( O(n^2) \), where \( n \) is the number of test cases within \( T \). The following text provides details.

WebTestRepair (Lines 1-29 of Algorithm 1) makes two calls in Lines 7-8; these calls are linearly bounded by the number of test cases \( n \) within a test suite \( T \). Because the while loop (Lines 11-25) makes \( I \) calls to DepRepair in Line 12, this loop is bounded by \( I^*O(n^2) \), where \( I \) is the number of rounds the algorithm is allowed to run and \( n \) is the number of test cases explored. Therefore, WebTestRepair is bounded by \( I^*O(n^2) \).

DepRepair (Lines 11-32 of Algorithm 2) contains two for loops (Lines 17-29) that are bounded by \( O(n^2) \), and a call to the RepairedList routine in Line 30, which is also bounded by \( O(n^2) \), because RepairedList (Lines 34-54 of Algorithm 2) contains two for loops. This renders the total complexity of DepRepair \( 2*I^*O(n^2) \), which is \( O(n^2) \).

Finally, because the number of rounds \( I \) is expected to be very small in practice (for example, the maximum number of rounds \( I \) needed in our study was five) it is bounded by the number of rounds the developer specifies.
IAENG International Journal of Computer Science, 49:2, IJCS_49_2_11

TABLE IV: Objects of Analysis

<table>
<thead>
<tr>
<th>App. Name</th>
<th>App. Version</th>
<th>App. LOC</th>
<th># Test Suites</th>
<th>Test Case LOC (Avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP-ADDRESSBOOK</td>
<td>7.0.0</td>
<td>14,610</td>
<td>20</td>
<td>55</td>
</tr>
<tr>
<td>MRBS</td>
<td>1.8.0</td>
<td>184,403</td>
<td>20</td>
<td>82</td>
</tr>
<tr>
<td>SCHOOLMATE</td>
<td>2.0.0</td>
<td>7,040</td>
<td>20</td>
<td>142</td>
</tr>
<tr>
<td>TIMECLOCK</td>
<td>1.04</td>
<td>19,560</td>
<td>20</td>
<td>90</td>
</tr>
<tr>
<td>YOURCONTACTS</td>
<td>2.2.2</td>
<td>32,944</td>
<td>20</td>
<td>63</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>5</strong></td>
<td><strong>243,100</strong></td>
<td><strong>100</strong></td>
<td><strong>432</strong></td>
</tr>
</tbody>
</table>

TABLE IV: Objects of Analysis

<table>
<thead>
<tr>
<th>App. Name</th>
<th>App. Version</th>
<th>App. LOC</th>
<th># Test Suites</th>
<th>Test Case LOC (Avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP-ADDRESSBOOK</td>
<td>7.0.0</td>
<td>14,610</td>
<td>20</td>
<td>55</td>
</tr>
<tr>
<td>MRBS</td>
<td>1.8.0</td>
<td>184,403</td>
<td>20</td>
<td>82</td>
</tr>
<tr>
<td>SCHOOLMATE</td>
<td>2.0.0</td>
<td>7,040</td>
<td>20</td>
<td>142</td>
</tr>
<tr>
<td>TIMECLOCK</td>
<td>1.04</td>
<td>19,560</td>
<td>20</td>
<td>90</td>
</tr>
<tr>
<td>YOURCONTACTS</td>
<td>2.2.2</td>
<td>32,944</td>
<td>20</td>
<td>63</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>5</strong></td>
<td><strong>243,100</strong></td>
<td><strong>100</strong></td>
<td><strong>432</strong></td>
</tr>
</tbody>
</table>

Fig. 1: An internal representation created for the command in Line 12 of Table III

Thus, I may be ignored and the computational complexity for WebTestRepair is O(n^2).

IV. Empirical Study

As previously stated in Section II in this work we focus on detecting and repairing test case dependencies that adversely affect test suites that have been re-ordered, such as by TCP techniques. To evaluate the performance of WebTestRepair in this context, we conducted an empirical study addressing the following research questions:

**RQ1: Effectiveness:** Given a prioritized test suite, T, in which test cases are broken due to test case dependencies, how effective is WebTestRepair at repairing T?

**RQ2: Efficiency:** Given a prioritized test suite, T, in which test cases are broken due to test case dependencies, how much does it cost to apply WebTestRepair to T?

A. Objects of Analysis

As objects of study we selected five open-source web applications that have been used in previous web testing research [2, 16, 17]. Table IV provides data on the web applications that we selected, including their names, versions, the number of (non-comment) lines of code (LOC) they contain as counted with [1], the number of test suites used in this study for each application, and the mean number (non-comment) lines of code for each test case.

PHP-ADDRESSBOOK [3] is a web-based address and phone book, contact manager, and an organizer. It supports groups, addresses, e-Mails, phone numbers and birthdays. MRBS [29] is a web application for multi-site booking of meeting rooms. SCHOOLMATE [34] is a school management web application. TIMECLOCK [38] is an employee management system that provides several functionalities such as creating offices, groups within offices, and user accounts, scheduling upcoming events, and managing employee’s sign-in sheets. YOURCONTACTS [40] allows users to manage contacts with names, emails and phone numbers. The applications are all written in PHP, and use JavaScript, HTML, MySQL, and CSS. To evaluate the performance of WebTestRepair on these web applications, we needed test suites that contain test case dependencies. We began by searching for publicly available test suites, but could not find any. Therefore, for each web application, we created a test suite that contains a set of test cases with dependencies using Selenium IDE [35] (Guarnieri et al. [13], in their study of a technique for test isolation for web applications, reported a similar difficulty, and followed a similar approach in their empirical study.) To do this, we performed a functionality analysis for each application to determine possible use-case scenarios. We used these use-case scenarios to record test cases. We ensured that these test cases all executed appropriately. The rightmost column in Table IV lists the numbers of test cases in the test suites created for each application.

B. Variables and Measures

1) Independent Variables: We consider one independent variable: manifest test dependency and repair technique. We utilize our new technique, WebTestRepair, and as a control we use a technique that randomly orders the test cases in a broken test suite in search for a test suite that does not break. The choice of a random approach as a control is necessitated by the absence of other existing execution-based techniques for detecting and repairing test case dependencies in web applications that handle the same test case structure we are using in this study, as explained in Section II. A similar choice was made in the study of DT Detector [41]—which does not target web applications—for similar reasons.

2) Dependent Variables: The effectiveness of a repair technique is simply a boolean measure: the technique succeeds or not. For efficiency, we measure the wall clock time required by the technique to detect and repair breakages.

C. Study Process

Each application comes with a test suite containing between 6-10 Java/Unit 4/ WebDriver Backed test cases. For each of these suites, we created an ordering such that all tests in each test suite pass without any test dependencies related failures (breakages). We call these suites, the original (unprioritized) test suites. For each of
these original test suites, we create 20 distinct randomly prioritized test suites as follows. For each original test suite, we first create a random ordering for this suite. Next, we execute this prioritized test suite in that random ordering to determine whether any tests break. If so, and if the ordering differs from any previously selected orderings, we selected the ordering as one that contains potential test dependencies. We repeated this process until we had obtained 20 distinct randomly prioritized orderings for each of original test suites.

We applied WebTestRepair to each of the 20 randomly prioritized test suites for each web application, paired with the original test suite. As noted in Section III, WebTestRepair may require multiple rounds to identify and repair dependencies. Our preliminary usage of the tool suggested, however, that five rounds were sufficient to detect and repair all dependencies; thus, we set the maximum number of rounds to five.

Where the random repair approach was concerned, for each of the 20 randomly prioritized test suites, we randomly selected a new ordering for the prioritized test suite, and then determined whether that ordering eliminated the breakage. Because the random technique does not have an inherent iteration limit we allowed it to run twice as long as the maximum time WebTestRepair needed to repair broken test suites for a given web application.

To answer RQ1, we counted the number of prioritized test suites for which each approach is able to produce a repaired test suite ordering such that, when executed in this order, all tests pass without any failures related to test dependency.. To answer RQ2, we measured the wall clock time used by the approaches, including the time required to perform detection and repair, and to execute the test suites. We gathered all data on a PC with an Intel Core i7-7500U @2.70 GHz CPU and 12 GB of RAM.

D. Results

Table V summarizes the performance WebTestRepair as compared to the random approach. Column 1 lists application names, Columns 2 and 3 list the numbers of broken test suites that WebTestRepair was able to repair and the percentages it repaired, respectively. Columns 4 and 5 list the numbers of broken test suites that the random approach was able to repair and the percentages it repaired, respectively. Column 6 shows the total time (in minutes) WebTestRepair required to analyze, repair, and execute broken tests for each application, including application setup time, algorithm exploration and repair time, and the time required to execute test cases on the application. Column 7 shows the total times (in minutes) that the random approach required to generate and execute broken tests for each application, including the time required to select different orderings and to execute test suites.

1) RQ1: Effectiveness: Columns 2–5 of Table V report data on effectiveness. As the data shows, WebTestRepair was able to detected and repaired 95 broken test suites and therefore achieving an overall success rate of 95%. The random repair approach was much less effective at repairing test suites even when granted more than twice the time WebTestRepair utilized; it produced repaired test suites only 11% of the time and for some applications, it was unable to repair any test suites at all, e.g., for the MRBS and TimeClock applications. In this study, WebTestRepair an 84% more effective in repairing broken test suites than the random approach.

2) RQ2: Efficiency: Columns 6–7 of Table V report data on efficiency. For WebTestRepair, Column 6 shows (for each web application) the mean time (in minutes) the approach required for the whole process to analyze, repair, validate and execute test suites for that application. For the random approach, Column 7 shows the total this approach required to generate and execute test suites. For example, while processing the MRBS application, WebTestRepair only required 459 minutes to achieve 95% success rate whereas the random approach spent doubled that time and was not able to generate a single test suite ordering in which all tests pass. In this study, the random approach spent more than 36 hours to only achieve 11% success rate, whereas WebTestRepair spent 16 hours to achieve 95% success rate. Therefore, WebTestRepair is much more efficient and effective than the random approach.

V. Discussion

To further investigate the performance of WebTestRepair across different applications, we provide additional data on WebTestRepair’s performance in Table VI. In this table, Columns 2–6 show that WebTestRepair analyzed 800 tests, identified 227 broken tests, detected 551 manifest test dependencies form which 438 were resolved by WebTestRepair, and only spent a total of 5.59 seconds to analyze and resolve test dependencies in order repair broken test suites. As Column 5 of Table VI shows, the time spent by WebTestRepair on analyzing all broken tests to detect and resolve manifest dependencies is a very small fraction of the total time reported earlier by Column 6 of Table V while the bulk of the time is consumed on executing the web applications. Recall from Section II that finding all of the dependencies in a test suite is an NP-Complete problem. Instead, WebTestRepair goal is to create a repaired test suite that runs without any dependency-related breakages: after such a test suite is found, WebTestRepair performs a confirmation run to assure the developer that a dependency-free suite has been produced and then terminates.

To further investigate the complexity of broken tests and how they may differ for different applications, we consider the number of broken tests encountered by WebTestRepair and the number of manifest test dependencies identified during this process. This data is shown in Columns 2–3 of Table VI. For example, consider the data provided for TimeClock and YourContacts applications. By looking at the number of test dependencies encountered by WebTestRepair during its processing of the TimeClock application, WebTestRepair resolved 178 manifest test dependencies in order to repair 84 broken tests, while WebTestRepair was required to only resolve a 26 manifest test dependencies in order to repair 26 broken tests for the YourContacts application. This indicates that test suites created for the TimeClock
application are more complicated and dependent on each other than those test suites created for the YourContacts application.

Figure 3 displays the distribution of broken tests and manifest test dependencies as they were identified by WebTestRepair across the 20 test suites used per application. In light of this data and the data reported in Table V and Table VI, note that the random approach performed poorly on all applications used in this study, whereas WebTestRepair was able to repair 95% of test suites across all applications. In particular, the random approach was only able to repair 35% of the test suites for YourContacts which has 26 broken tests with 26 test dependencies. For the MRBS application, however, with 63 broken tests and 145 test dependencies, and TimeClock with 84 failed tests and 217 test dependencies, the random approach failed to repair any test suites.

As a further look, we considered the progress WebTestRepair makes across iterations. When WebTestRepair is invoked with a test suite that contains broken test cases, the algorithm builds the repaired test suite incrementally through iterations, during which the partially repaired suite produced by a previous round is forwarded to the next round for repair, and so on, until a repaired test suite is produced or WebTestRepair reaches the maximum prescribed number of rounds. If the test suite being processed is not fully repaired during an iteration, WebTestRepair logs the resultant partially repaired test suite and passes this suite to the next round where it is processed as if it were a new one.

While considering a broken test suite, WebTestRepair may encounter different numbers of broken test cases during each iteration. The primary reasons this number changes is related to how many test cases are truly dependent on each other; repairing existing dependencies may introduce new ones.

In this work we chose not to consider the number of iterations performed by WebTestRepair as an independent variable. Instead, we chose a fixed number of iterations (five). We did this because preliminary runs suggested that this number of iterations was sufficient for the web applications considered, and this turned out to be the case when we proceeded with the study. To further understand the progress WebTestRepair makes across iterations, we present data related to the iterations WebTestRepair performed when processing the twenty test suites for TimeClock in Table VII.

Column 1 of Table VII lists test suites being processed. Columns 2–6 report the number of test case breakages that WebTestRepair detected during each of the iterations it conducted while processing each test suite. For example, while processing test suite 1, WebTestRepair detected six broken test cases, but the algorithm required just one iteration to produce a repaired test suite. While processing test suite 4, in contrast, WebTestRepair required five iterations. In the first iteration, WebTestRepair identified six broken test cases and was able to repair five of them, leaving one unrepaired. It thus performed a second iteration in which it tried to repair the one test case breakage that still remained. The effect of this iteration, however, was to produce additional test case breakages. In its third iteration, WebTestRepair detected four broken test cases produced by the second round and attempted to repair them, and this resulted in a test suite containing one broken test case. WebTestRepair continued to attempt to repair this test suite in its fourth iteration, and this attempt produced a test suite with five broken test cases that were forwarded to the fifth iteration. During that final iteration, WebTestRepair succeeded in repairing all test case breakages.

In practice, when considering different web applications and test suites, the number of iterations required may differ, and developers may choose to utilize different numbers of iterations based on both cost and effectiveness factors. Appropriate numbers could be determined over time as regression testing is performed on successive releases.
VI. Related Work

Zhang et al. [41] analyze test case dependence history data obtained from 96 test cases collected from five software issue tracking systems and propose four heuristic algorithms for detecting test case dependencies within these test cases. Zhang et al. implement a test case dependence detector as a tool called DTDetector. The approach has three potential drawbacks. First, it requires that history information about a test suite be retrieved from developers, and this may be difficult or impossible in practice. Second, the approach is not fully automated and requires quite a bit of human interaction [7]. A third problem relates to the intractability of the problems addressed by the proposed algorithms, which render DTDetector relatively expensive. WebTestRepair, in contrast, does not perform exhaustive searches and is bounded in terms of execution time; it concentrates on producing repairs only to the test suite breakages caused by test case dependencies.

Bell et al. [7] present a tool called ElectricTest, which utilizes built-in dynamic Java capabilities and resources such as IO-Traces, garbage collection, profiling, and the JVM Tooling Interface (JVMTI) to detect test case dependencies created by read/write operations to shared global resources such as memory, files, and the network. In [12], PRADET is presented as an improvement to ElectricTest [7] and DTDectector [41]. PRADET focuses on state polluting tests [14] and therefore, it detects data dependencies at this stage by capturing each read after write (RAW) operations performed on static global objects in Java projects. As an improvement over the work of ElectricTest and DTDectector, PRADET performs a refinement operation to report only manifest test dependencies as opposed to the final list reported by ElectricTest and DTDectector which includes all test dependencies found in a test suite.

All of DTDectector [41], ElectricTest [7], and PRADET [12], at first perform a very extensive search for all test dependencies and do not provide any repair process. WebTestRepair, in contrast, follows an...
languages such as HTML, JavaScript, CSS, in addition to sever-side code. Web application code may be composed of multiple languages such as HTML, JavaScript, CSS, in addition to sever-side code.

Lam et al. [24] extend the work of [41] to cope with test case dependencies when they exist; however, this work inherits the drawbacks of the approach reported in [41], noted above. WebTestRepair focuses on automatically improving test suites through the process of test case dependence detection, repair, and then confirmed execution.

The work of [6, 13, 14] recognize the existence of the test dependency problem and how the side-effects created by such dependencies may negatively affect testing results. To reduce these problems they propose techniques for test case dependency prevention and test case execution optimization. Bell [6] proposes a technique for test case dependency identification and then uses this knowledge to improve test case execution efficiency.

Gyori et al. [14] present a technique (PolDet) that exposes shared memory locations that may have negative side-effects when conducting test runs on Java programs. Guarneri et al. [13] propose a technique called Test Execution Checkpointing (TEC), which includes a framework for running tests in isolation. This work targets web applications, but focuses solely on server side tests sent as HTTP requests, whereas WebTestRepair handles tests of both client-side and server-side code. The foregoing techniques all avoid test dependency side-effects during testing through prevention, by running test cases in isolation, whereas WebTestRepair focuses on detection and repair.

Alshahwan and Harman [4] present a technique for repairing web applications session data. This work analyzes web application session data in order to detect and repair session paths that may be broken because of changes made to a web application. Although this approach applies the same philosophy of analysis and repair as WebTestRepair, the approach focuses on the analysis of web application session data in the form of URL requests (server-side), whereas our approach analyzes and repairs web applications test case code that may be created using multiple languages such as HTML, JavaScript, CSS, in addition to sever-side code.

Haidry and Miller [15] present a TCP technique that functions in the presence of test case dependencies. However, this work does not provide any technique for detecting dependencies and assumes they are known in advance. The test suites produced by WebTestRepair could be utilized as input to the technique proposed in [15]. The work in [23] presents an algorithm that regenerates locators by saving extra information from the previous structure of the website. Although this approach is promising, it could not be used for the purpose of test dependencies detection and repair because the work of WebTestRepair is mainly based on the dynamic changes created by the test’s code during its execution and not on an old website structure.

Sung et al. [37] use DOM tree [9] analysis of JavaScript code to identify event handlers’ dependencies in web applications. This work considers those events that are triggered by test cases and shows that event dependencies lead to test case dependencies; however, the work considers only event handlers, whereas our work takes a broader view of web applications and considers test case dependencies that are related to the entire functionality of web applications.
of the web application. Our work, in contrast, does not require analysis of the application’s code; it works only with test code, and thus can be more efficient.

Biagiola et al. [8] presents a tool called TEDD, for test dependency detection in E2E web tests. TEDD follows the same methodology reported in PRADET [12] to produce a list of manifest test dependencies. However, as an improvement, TEDD adds a test recovery step after the refinement stage, used in PRADET, in order to recover some dependencies that me be wrongfully dropped during the refinement process. During its work, TEDD starts with a test suite and a given original execution ordering and performs data-flow analysis on the test code to search for test data dependencies and therefore build a test dependency graph (TDG) will all test dependencies. As in PRADET, TEDD refines the resulting TDG by keeping only manifest dependencies. As stated in [8], some manifest test dependencies might be wrongfully removed during the refinement stage, therefore, TEDD performs an extra step during which lost dependencies are recovered. WebTestRepair, in contrast, from the outset, focuses on the detection of manifest test dependencies that cause test failures.

Additionally, TEDD assumes the existence of meaningful test names during the application of its natural language processing (NLP) analysis for test dependency detection. Therefore, the effectiveness and practicality of TEDD is limited to web application tests with meaningful names, and may become inapplicable in many real-life web applications with tests that do not adhere to this assumption.

Further, during its test code analysis, TEDD assumes the existence of assertions in test’s code, ( [8], p. 2), and then limit its test dependency search to Selenium’s “assert” statements. This assumption crates two drawbacks potential drawbacks with TEDD. First, it enforces developers to write assertions, and writing good assertions is not an easy task and may add an extra burden. Second, this assumption makes assertions to be the sole source of test dependency, but this assumption is not valid in practice, because any other statement in a web app test may causes test dependency if this statement manipulates a shared web-page element.

In our work, WebTestRepair considers all web test statements, including assertions, during its search for test dependencies, and, therefore covers a wider spectrum of web tests, which makes it more practical than TEDD. Further, TEDD does not provide any test dependency repair and only reports test dependencies, while WebTestRepair provides both test dependency detection and repair.

VII. CONCLUSION

We have presented a novel algorithm, WebTestRepair, for automatically detecting and repairing manifest dependencies in web application test suites, along with an implementation of that algorithm. We evaluated both the effectiveness and efficiency of WebTestRepair on five non-trivial web applications. Across 800 tests of these web applications, WebTestRepair was able to detect manifest test dependencies and produce repaired test suites that run without dependency-related failures with a 95% success rate. In the future, we intend to extend WebTestRepair to include automated DOM traversal techniques in order to improve the process of web element identification during the search for manifest test dependencies. Finally, we intend to extend our tool to handle different forms of Selenium test case exports, and study its applicability on those.

References


[34] Schoolmate. https://sourceforge.net/projects/schoolmate/.


