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 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].

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).

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. Such scenarios have motivated research on detecting and coping with test dependency [7], [8], [12], [21], [24], [41].

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 [41], 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 DTDectector 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

order to only report manifest test dependencies to the developer. All of DTDectector, 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 [7], [8], [12], [41]. For example, during their work with shared global variables in Java projects, [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 DTDectector [41], ElectricTest [7] and PRADET [12] are not applicable in our context.

Similarly, TEDD [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 ([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 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 [12], WebTestRepair performs a combined string and data-flow analysis. As compared to exiting work reported in [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 refereed to in [41] as "Dependent Test Detection Problem," because we are interested in finding only those dependencies that *cause observable test 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 [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 II provides background information. Section III presents WebTestRepair and an example of its use. Section IV presents our empirical study and results. In Section V we discuss our results and their implications. Section VI describes related work and Section VII presents conclusions and future work.

II. Background

As noted in Section I, this work focuses on tests created using Selenium IDE [35]. Selenium IDE allows developers to record their interactions with a web application.

TABLE I: A Test Suite Specification for the Schoolmate App

Test	Functionality
t_1	Add a user account of type "Admin", three accounts of type "Teacher" and one account of type "Student".
t_2	Fill in personal information for all user accounts.
t_3	Create two terms and in each term create a semester.
t_4	Add a new user account of type "Parent" and update school information.
t_5	Add two new classes.
t_6	Delete a term.

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 $T' = (t_4, t_2, t_3, t_5, t_6, t_1)$. 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 III-C 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 III-B provides a full description of WebTestRepair, and in Section III-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], [41], 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_b 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_b breakage. In contrast, prior work [7], [8], [12], [21], [24], [41], 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 failures, 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(read/write/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)

Algorithm 1 WebTestRepair Algorithm – Main Routine

```
1: procedure WebTestRepair (A, T_o, T_m, I): T_r
2:     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_o ← 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_r ← DepRepair(traceT_o, TraceT_m)
13:        if (T_r is empty) then
14:            exit while loop
15:        else
16:            if (all test cases in T_r 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 find repaired suite
28:    end if
29: end procedure
```

Algorithm 2 WebTestRepair Algorithm - Primary Subroutines

```
1: procedure PreProcess (A, T): traceT
2:     A: web application
3:     T: a test suite for A
4:     traceT: trace for T on A
5:     preT ← Prepare(T)
6:     T̂ ← Instrument(preT)
7:     traceT ← Execute(A, T̂)
8:     return traceT
9: end procedure
10:
11: procedure DepRepair (traceT_o, traceT_m): T_r
12:     traceT_o: execution trace for original test suite
13:     traceT_m: execution trace for broken test suite
14:     T_f: a repaired version of the broken test suite
15:     bTests ← AllBroken(traceT_m)
16:     allBdeplists ← empty
17:     for (each test case t_b in bTests) do
18:         oTtrace ← GetTrace(t_b, traceT_o)
19:         pS ← ProblemS(t_b, oTrace)
20:         index ← Location(oTrace)
21:         bDeplist ← empty
22:         bDeplist ← AddFirst(t_b)
23:         for (each test case t_i in traceT_o backwards from index) do
24:             if (Referenced(pS, t)) 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:
34: 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 excT_m backwards from tail) do
40:         if (Broken(t_i) then
41:             bDeplist ← dependence list of t_i from allBdeplists
42:             for (each test case t_d in bDeplist backwards from tail)
    do
43:                 if (t_d is not in T_f) then
44:                     T_f ← AddFirst(t_d)
45:                 end if
46:             end for
47:         else
48:             if (t_i is not in T_f & t_i is not in bDeplist) then
49:                 T_f ← AddFirst(t_i)
50:             end if
51:         end if
52:     end for
53:     return T_f
54: end procedure
```

data-flow dependency between t_i and t_b, $(t_i \rightarrow t_b)$, if t_b tries to reference (read, write, or delete) a web-page element that has been deleted by t_i. A (RbC) data-flow dependency exists between t_i and t_b, $(t_i \rightarrow t_b)$, if t_b tries to reference (read, write, or delete) a web-page element which yet to be created by t_i.

To explain the difference between RaD and RbC test data-flow dependencies, consider the Schoolmate web application's test suite specification introduced previously in I. For example, if t_2 tries to fill a personal details webform 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_z, tries to reference (read/write/delete) a term that was created before by t_3 but was later deleted by t_6, then there exits a RaD data-flow dependency between t_6 and t_z $(t_6 \rightarrow t_z)$.

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_r, 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 round, a list of test dependency that may caused a certain test to fail, a detailed description of code's snippet that may caused test dependency, and the number of tests inspected during the search for test dependencies. Statistics files include: the time required for test dependency detection, time required for test dependency repair, time required to execute the application, and the number of rounds required to produce a solution.

WebTestRepair starts its work in (Lines 7–8) by making two consecutive calls to a prepossessing routine

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): T̂
7:     Input: T, a test suite
8:     Output: T̂, an instrumented version of T
9: end procedure
10:
11: procedure EXECUTE (A, T): traceT
12:     Input: A, a web application
13:            T, a test suite for A
14:     Output: trace, an execution trace for T on A along with execution
15:             results for all test cases in T
16: end procedure
17:
18: procedure ALLBROKEN (traceT_m): bTests
19:     Input: traceT_m, the execution trace for the broken test suite
20:     Output: bTests, a list of broken test cases in T_m
21: end procedure
22:
23: procedure GETTRACE (t_b, traceT_o): oTtrace
24:     Input: t_b, a broken test case
25:            traceT, the execution trace for the broken test suite
26:     Output: oTtrace, the execution trace of t_b on the original test suite
27: end procedure
28:
29: procedure PROBLEMS (t_b, oTrace): pS
30:     Input: t_b, a broken test case
31:            oTrace, the execution trace of t_b on the original test suite
32:     Output: pS, the statement in t_b that caused t_b to break
33: end procedure
34:
35: procedure REFERENCED (pS, t_i):result
36:     Input: pS, the statement in test case t_i that caused t_i to break
37:            t_i, a broken test case
38:     Output: result, true if pT was read, written, or modified by t_i, false otherwise
39: end procedure
40:
41: procedure BROKEN (t_i):result
42:     Input: t_i, a test case
43:     Output: result, true if t_i is broken, false otherwise
44: end procedure
```

called PreProcess. The purpose of these two calls is to preprocess the original test suite T_o and its prioritized version T_m, respectively. PreProcess ultimately returns an execution trace for T_o and T_m. 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 T_m 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.

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 T_o and T_m, 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 II shows a Selenium generated Java/Unit4/WebDriverBacked code for test (t_1) 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 includes 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, T_o and T_m, the PreProcess invokes the Execute routine (Algorithm 3) to executes 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= $\{c_1, c_2, \ldots c_n\}$, for each command $c_k \in C$, we create a data-structure $cmdRep(c_k)$= <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 abut surrounding commands of c_k within the same test T. Figure 1 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

TABLE II: Java/Unit 4/WebDriver/Backed code exported from Selenium-IDE for Test Case t_1

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

(*WaitForPageToLoad* and *sotp*) because they 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 $cmdRep(c_k)$ 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 $traceT_o$ and $traceT_m$, the execution traces for T_o and T_m, respectively. DepRepair performs a combined string and data-flow analysis on these execution traces to identify test dependencies that caused test breakages in T_m. After identifying test dependencies, DepRepair uses T_o as an oracle and starts a repair process for T_m trying to produce a working execution ordering for T_m such that when executed in this order all tests pass. Note that $traceT_o$ contains *all pass* execution trace, while $traceT_m$ 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, $traceT_o$ and $traceT_m$, 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 $traceT_m$. Next, in the `for` loop of Lines 17–29, DepRepair considers each broken test in $bTests$. For each broken test t_b, DepRepair calls GetTrace (Algorithm 3) to obtain a *pass* execution trace for t_b from $traceT_o$ which will be utilized later to locate the breakage's location in t_b.

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

To identify manifest test dependencies causing the breakage of test t_b, DepRepair considers each test t_i that was executed *before* t_b from the original test suite execution trace, $traceT_o$, as shown by the code in the `for` loop of Lines 23–27. Specifically, for each t_i, 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 t_i and t_b.

The Referenced procedure begins the process of manifest test dependency detection by performing a string-analysis searching each test t_i for clues that may lead to locating any links between the shared web element that caused t_b 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 RbC) between t_i and t_b, using the data-flow dependency classification defined in Section III-A. This extra step in necessary because a shared web page element between t_b and t_i 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 tests 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 logarithm 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. Eventfully, 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 $traceT_m$, 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 II. These test suits are: $T_o = (t_1, t_2, t_3, t_4, t_5, t_6)$ and its prioritized version $T_m = (t_4, t_2, t_3, t_5, t_6, t_1)$. Recall, from Section II, that these tests are created based on the specification of Table I, and that both t_2 and t_5 failed during the execution of T_m due to test dependency.

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 eventfully 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_4, t_2, t_3, 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 III, 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. DepRepaire also needs to know those tests that were executed before t_2 in suite T_o. For our example, DepRepaire 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, DepRepaire 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 II. 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. $bDeplist = <t_1 \rightarrow t_2>$. Although there was only a 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 $bDeplist = <t_2 \rightarrow t_5>$. Eventually, as described in Section III-B, DepRepair adds this newly generated dependency list to $allBdeplists$, 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

Line	Code
1	selenium.open("/schoolmateTest/index.php");
2	selenium.type("name=username", "test");
3	selenium.type("name=password", "test");
4	selenium.click("css=input[type='submit']");
5	selenium.waitForPageToLoad("60000");
6	selenium.click("link=Teachers");
7	selenium.waitForPageToLoad("60000");
8	selenium.click("xpath=(//input[@value='Add'])[2]");
9	selenium.waitForPageToLoad("60000");
10	selenium.type("name=fname", "Lisa");
11	selenium.type("name=lname", "Harry");
12	*selenium.select("name=username", "label=u2");*
13	*selenium.click("css=input[type='button']");*
14	*selenium.waitForPageToLoad("60000");*
15	*selenium.click("xpath=(//input[@value='Add'])[2]");*
16	*selenium.waitForPageToLoad("60000");*
17	*selenium.type("name=fname", "Sara");*
18	*selenium.type("name=lname", "Cameron");*
19	*selenium.select("name=username", "label=u1");*
20	*selenium.click("css=input[type='button']");*
21	*selenium.waitForPageToLoad("60000");*
22	*selenium.click("xpath=(//input[@value='Add'])[2]");*
23	*selenium.waitForPageToLoad("60000");*
24	*selenium.type("name=fname", "Moody");*
25	*selenium.type("name=lname", "Aley");*
26	*selenium.select("name=username", "label=moody");*
28	*selenium.waitForPageToLoad("60000");*
29	*selenium.click("link=Log Out");*
30	*selenium.waitForPageToLoad("60000");*

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

Next, t_6 is considered and 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 is 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 $T_f = <t_4, t_1, t_3, t_2, t_5, t_6>$. 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 $T_m = (t_4, t_1, t_3, t_2, t_5, t_6)$ 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 [26] 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 it 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*$O(n^2)$, 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.

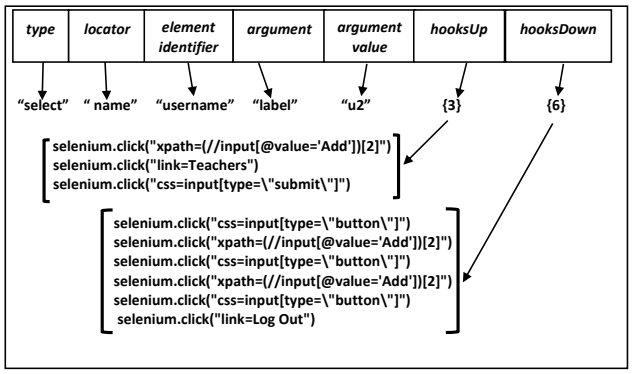

Fig. 1: An internal representation created for the command in Line 12 of Table III.

Line	Code
1	selenium.open("/addressbookv7.0.0/index.php");
2	**selenium.click("link=groups");**
3	selenium.waitForPageToLoad("30000");
4	**selenium.click("xpath=(//input[@name='selected[]'])[2]");**
5	selenium.click("xpath=(//input[@name='edit'])[2]");
6	selenium.waitForPageToLoad("30000");
7	selenium.type("name=group_header", "Welcome to Students group");
8	selenium.type("name=group_footer", "It was our pleasure serving you!");
9	**selenium.click("name=update");**
10	selenium.waitForPageToLoad("30000");

Fig. 2: An example of searching surrounding code of a failed command during test dependency detection.

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

IV. Empirical Study

As previously stated in Section I, 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]scs 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.

TABLE IV: Objects of Analysis

App. Name	App. Version	App. LOC	# Test Suites	Test Case LOC (Avg)
PHP-AddressBook	7.0.0	14,610	20	55
MRBS	1.8.0	184,403	20	82
Schoolmate	2.0.0	7,040	20	142
TimeClock	1.04	19,560	20	90
YourContacts	2.2.2	32,944	20	63
Total 5		243,947	100	432

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 tests dependencies in web applications that handle the same test case structure we are using in this study, as explained in Section I. A similar choice was made in the study of DTDetector [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 I 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

TABLE V: Study Results

	Repair Effectiveness				Repair Efficiency	
Application Name	**WebTestRepair**		**Random**		**WebTestRepair**	**Random**
	#Suites Repaired	**Repair %**	**# Suites Repaired**	**Repair %**	**Costs (minutes)**	**Costs (minutes)**
php-AddressBook	18	90%	2	10%	145	301
MRBS	19	95%	0	0%	459	920
Schoolmate	20	100%	2	10%	219	446
TimeClock	18	90%	0	0%	97	427
YourContacts	20	100%	7	35%	45	73
Total	95	95%	11	11%	965	2,167

TABLE VI: WebTestRepair Performance

Web App Name	**# Tests**	**# Identified Broken Tests**	**# Detected Manifest Dependencies**	**# Resolved Manifest Dependencies**	**Total Analysis Cost (Seconds)**
php-AddressBook	200	26	117	59	1.16
MRBS	180	63	145	129	1.72
Schoolmate	120	28	46	46	0.72
TimeClock	160	84	217	178	1.64
YourContacts	140	26	26	26	0.39
Total	800	227	551	438	5.63

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 suites 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.

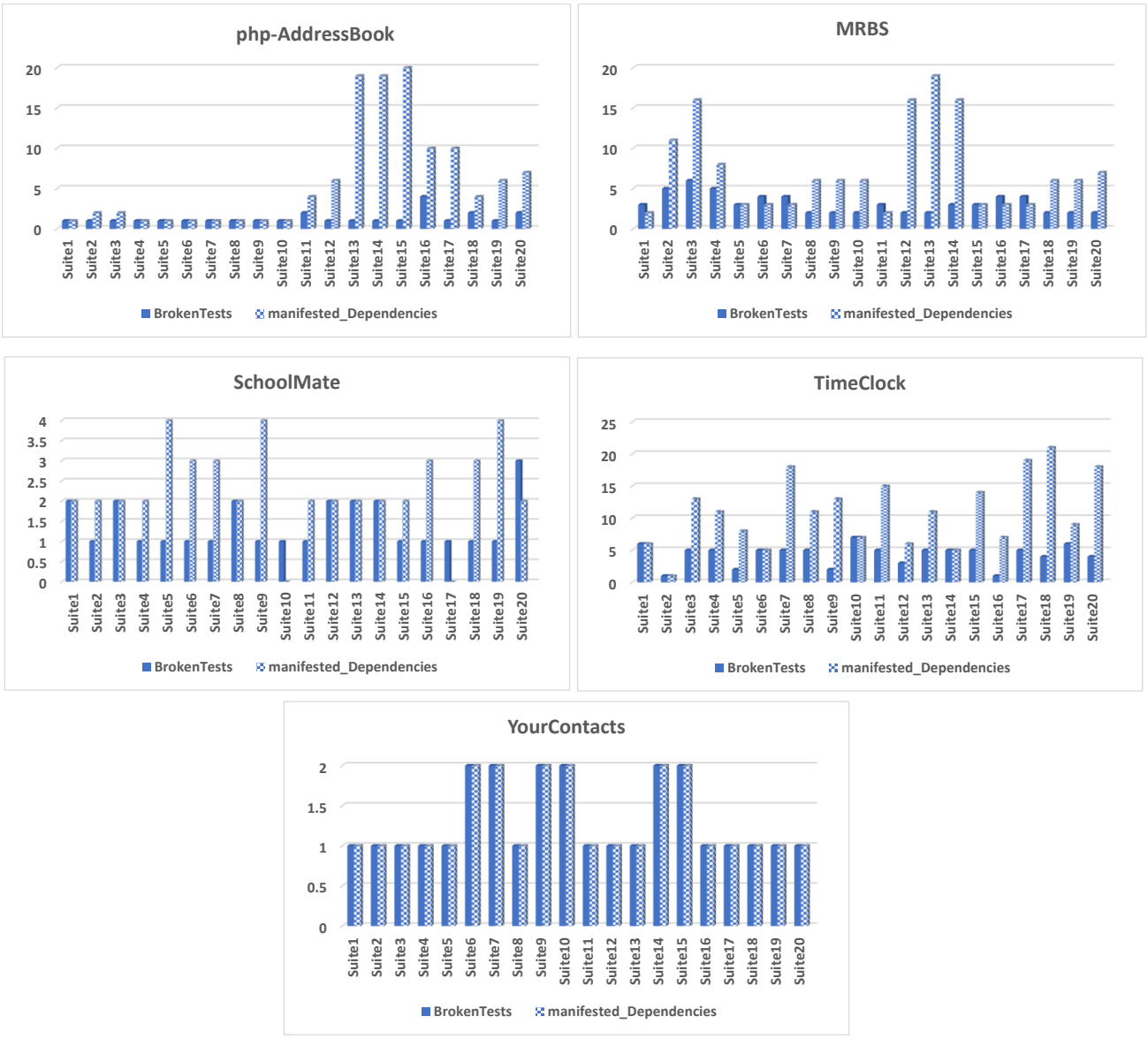

Fig. 3: The distribution of broken tests and manifest test dependencies identified by WebTestRepir across all applications.

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

TABLE VII: Dependencies that Caused Test Case Breakages in TimeClock

	Number of Broken Test Cases				
Suite #	**R1**	**R2**	**R3**	**R4**	**R5**
1	6	-	-	-	-
2	3	6	4	5	-
3	3	5	6	4	4
4	6	1	4	1	5
5	2	3	1	2	-
6	5	6	-	-	-
7	4	2	3	4	5
8	6	-	-	-	-
9	2	5	5	-	-
10	7	-	-	-	-
11	3	3	4	5	-
12	3	3	-	-	-
13	6	6	-	-	-
14	5	-	-	-	-
15	5	5	6	4	5
16	6	2	-	-	-
17	5	2	3	4	5
18	7	2	3	4	5
19	3	6	-	-	-
20	4	6	4	5	-

optimistic greedy and incremental approach to detect the smallest number of dependencies that need to be repaired to produce a dependence-free test suite. This renders our approach more suitable for developers who are interested only in manipulating test suites without identifying all test case dependencies. TCP, e.g., [11], [19], [32], [36] techniques may safely be applied to the test suites produced by WebTestRepair.

Kappler [21] attempts to speed up test case parallelization in the presence of test case dependencies, and reduce the overheard associated with the parallelization technique of [7]. The main objective of [21] is to improve the performance of parallelized test case execution using techniques for coping with test case dependencies. Our work, in contrast, focuses on improving the applicability of test case manipulation techniques such as TCP in the context of regression testing of web applications. Furthermore, all of prior work presented in [7], [12], [21], [41] provide dependence detection mechanisms but do not perform dependence repair; however, WebTestRepair provides both.

WebTestRepair faces more challenges that are not faced by DTDectector [41], ElectricTest [7], and PRADET [12]. One of these issues involves the deletion of objects. Both DTDectector and ElectricTest trace read/write operations to objects and consider test B to be dependent on test A only if B reads an object that was written by A. WebTestRepair targets a different software paradigm, and faces different challenges than tools concerned with traditional programs that use variables, I/O and network resources within the context of a test case. 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. Guarnieri 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 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 dependence 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 elements 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

[1] Cloc 2019. Counts blank lines, comment lines, and physical lines of source code in many programming languages. https://github.com/AlDanial/cloc, 2019.

[2] TEDD 2019. Web test dependency detection using nlp. https://github.com/matteobiagiola/FSE19-submission-material-TEDD, 2019.

[3] Addressbook. https://sourceforge.net/projects/php-addressbook/.

[4] N. Alshawan and M. Harman. Automated session data repair for web application regression testing. In *Proceedings of the International Conference on Software Testing*, 2008.

[5] Maral Azizi. A tag-based recommender system for regression test case prioritization. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 146–157, 2021.

[6] J. Bell. Detecting, isolating and enforcing dependencies among and within test cases. In *Proceedings of the International Symposium on Foundations of Software Engineering*, 2014.

[7] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya. Efficient dependency detection for safe Java test acceleration. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 770–781, 2015.

[8] Matteo Biagiola, Andrea Stocco, Ali Mesbah, Filippo Ricca, and Paolo Tonella. Web test dependency detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 154–164, 2019.

[9] Document object model (dom). http://www.w3.org/.

[10] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 253–264, 2006.

[11] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 102–112, 2000.

[12] Alessio Gambi, Jonathan Bell, and Andreas Zeller. Practical test dependency detection. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–11. IEEE, 2018.

[13] M. Guarnieri, P. Tsankov, T. Buchs, M. T. Dashti, and D. Basin. Test execution checkpointing for web applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2017.

[14] A. Gyori, A. Shi, F. Hariri, and D. Marinov. Reliable testing: Deteting state-polluting tests to prevent test dependencey. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2015.

[15] S.-e-Z. Haidry and T. Miller. Using dependency structures for prioritization of functional test suites. *IEEE Transactions on Software Engineering*, 39(2):258–275, February 2013.

[16] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. Waterfall: An incremental approach for repairing record-replay tests of web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 751–762, 2016.

[17] Mouna Hammoudi, Gregg Rothermel, and Paolo Tonella. Why do record/replay tests of web applications break? In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 180–190. IEEE, 2016.

[18] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 312–326, 2001.

[19] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *Proceedings of the International Conference on Automated Software Engineering*, pages 233–244, 2009.
[20] Jeonghyun Joo, Seunghoon Yoo, and Myunghwan Park. Poster: Test case prioritization using error propagation probability. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 398–401, 2020.
[21] S. Kappler. Finding and breaking test dependencies to speed up test execution. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 1136–1138, 2016.
[22] T. Kim, R. Chandra, and N. Zeldovich. Optimizing unit test execution in large software programs using dependency analysis. In *Proceedings of the Asia-Pacific Workshop on Systems*, pages 19:1–19:6, 2013.
[23] Hiroyuki Kirinuki, Haruto Tanno, and Katsuyuki Natsukawa. Color: Correct locator recommender for broken test scripts using various clues in web application. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 310–320, 2019.
[24] W. Lam, S. Zhang, and M. D. Ernst. When tests collide: Evaluating and coping with the impact of test dependence. Technical Report UW-CSE-15-03-01, University of Washington Department of Computer Scieence and Engineering, Seattle, WA, March 2015.
[25] Xiao Ling, Rishabh Agrawal, and Tim Menzies. How different is test case prioritization for open and closed source projects. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.
[26] Mamp. https://www.mamp.info/en/.
[27] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov. Parallel test generation and execution with Korat. In *Proceedings of the Symposium on The Foundations of Software Engineering*, pages 135–144, 2007.
[28] Shouvick Mondal and Rupesh Nasre. Summary of hansie: Hybrid and consensus regression test prioritization. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 278–280, 2021.
[29] Mrbs. http://sourceforge.net/projects/php-addressbook/.
[30] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso. Regression testing in the presence of non-code changes. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 21–30, 2011.
[31] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 241–251, 2004.
[32] M. J. Rummel, G. M. Kapfhammer, and A. Thall. Towards the prioritization of regression test suites with data flow information. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1499–1504, 2005.
[33] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *Proceedings of the International Conference on Automated Software Engineering*, pages 114–123, 2005.
[34] Schoolmate. https://sourceforge.net/projects/schoolmate/.
[35] Selenium hq. http://seleniumhq.org/.
[36] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the ACM Symposium on Applied Computing*, pages 97–106, 2002.
[37] C. Sung, M. Kusano, N. Sinha, and C. Wang. Static DOM event dependency analysis for testing web applications. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 447–459, 2016.
[38] Timeclock. https://sourceforge.net/projects/timeclock/.
[39] M. Wu, F. Long, X. Wang, Z. Xu, H. Lin, X. Liu, Z. Guo, H. Guo, L. Zhou, and Z. Zhang. Language-based replay via data flow cut. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 197–206, 2010.
[40] Yourcontacts. https://github.com/jubi4dition/yourcontacts.
[41] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 385–396, 2014.

Ali M. Alakeel previously known as Ali M. Al-Yami, obtained his PhD degree in computer science from Illinois Institute of Technology, Chicago, USA in Dec. 1996, his M.S. degree in computer science from the University of Western Michigan, Kalamazoo, USA in Dec. 1992 and his B.Sc. degree in computer science from King Saud University, Riyadh, Saudi Arabia in Dec. 1987. He is currently a professor of computer science at the University of Tabuk, Saudi Arabia. His current research interests include automated software testing, artificial intelligence, fuzzy logic and distributed computing.