Web Test Dependency Detection

Matteo Biagiola[*]
Fondazione Bruno Kessler
Trento, Italy
biagiola@fbk.eu

Andrea Stocco
Università della Svizzera Italiana
Lugano, Switzerland
andrea.stocco@usi.ch

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

Filippo Ricca
Università degli Studi di Genova
Genoa, Italy
filippo.ricca@unige.it

Paolo Tonella
Università della Svizzera Italiana
Lugano, Switzerland
paolo.tonella@usi.ch

ABSTRACT

E2E web test suites are prone to test dependencies due to the heterogeneous multi-tiered nature of modern web apps, which makes it difficult for developers to create isolated program states for each test case. In this paper, we present the first approach for detecting and validating test dependencies present in E2E web test suites. Our approach employs string analysis to extract an approximated set of dependencies from the test code. It then filters potential false dependencies through natural language processing of test names. Finally, it validates all dependencies, and uses a novel recovery algorithm to ensure no true dependencies are missed in the final test dependency graph. Our approach is implemented in a tool called TEDD and evaluated on the test suites of six open-source web apps. Our results show that TEDD can correctly detect and validate test dependencies up to 72% faster than the baseline with the original test ordering in which the graph contains all possible dependencies. The test dependency graphs produced by TEDD enable test execution parallelization, with a speed-up factor of up to 7×.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**;

KEYWORDS

web testing, test dependency, NLP

ACM Reference Format:
Matteo Biagiola, Andrea Stocco, Ali Mesbah, Filippo Ricca, and Paolo Tonella. 2019. Web Test Dependency Detection. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338948

1 INTRODUCTION

Ideally, all tests in a test suite should be independent. However, in practice, developers create tests that are *dependent* on each other [2, 12, 17, 21, 42]. Test dependency can be informally defined as follows. Let $T = \langle t_1, t_2, \ldots, t_n \rangle$ be a test suite, where each t_i is a test case, whose index i defines an order relation between test cases that corresponds to the original execution order given by testers. When tests within T are executed in the original order, all tests execute correctly. If the original execution ordering is altered, e.g., by executing t_2 before t_1, and the execution of t_2 fails, we can say that t_2 *depends on* t_1 for its execution, and that a *manifest test dependency* exists [12, 42].

Test dependencies inhibit the use of test optimization techniques such as test parallelization [1], test prioritization [33], test selection [14] and test minimization [39], which all require having independent test cases. Furthermore, test dependencies can mask program faults and lead to undesirable misleading side-effects, such as tests that pass when they should fail, tests that fail when they should pass [23, 42], or significant test execution overheads [2, 21].

Web developers frequently use end-to-end (E2E) test automation tools, which verify the correct functioning of the application in given test scenarios by means of automated test scripts. Such scripts automate the manual operations that the end user would perform on the web application's graphical user interface (GUI), such as delivering events with clicks, or filling in forms [5–7, 11, 13, 18, 26–28, 37]. Unlike traditional unit tests, E2E tests focus on whole business-oriented scenarios, such as logging into the web application, adding items to a shopping cart, and checking out. Such test scenarios go through all tiers and involve all services required to make the whole application work. Thus, in web testing, it can be difficult to enforce isolation, as web tests might use the application state which is promptly available from previous test case executions (i.e., a polluted state [17]). This creates potential test dependencies due to read-after-write operations performed on persistent data such as database records or Document Object Model (DOM) fragments that are written by a test t_i and later accessed by a successive test t_j (where $j > i$).

Automated detection of all test dependencies in any given test suite is NP-complete [42]. As such, researchers have proposed techniques and heuristics that help developers detect an approximation of such dependencies in a timely manner [2, 12, 17, 21, 42]. In this work, we focus on automatically detecting test dependencies in E2E web tests. Existing tools such as DTDetector [42], ElectricTest [2]

[*] Also with Università degli Studi di Genova.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338948

or PRADET [12] are not applicable because they are based on the extraction of read/write operations affecting shared data (e.g., static fields) of Java objects. Instead, web applications are prone to dependencies due to the persistent data managed on the server-side and the implicit shared data structure on the client-side represented by the DOM. Such dependencies are spread across multiple tiers/services of the web application architecture and are highly dynamic in nature. Hence, existing techniques based on static code analysis are not directly applicable. The web test dependency problem demands for novel approaches that leverage the information available in the web test code and on the client side.

In this paper, we propose a novel test dependency detection technique for E2E web test cases based on *string analysis* and *natural language processing* (NLP). Our approach is implemented in a tool called TEDD (**Te**st **D**ependency **D**etector), which supports efficient and conservative detection and validation of test dependencies in an E2E test suite using only client-side information, which makes it independent of the server-side technology.

TEDD *extracts* an initial approximated dependency graph (TDG). Then, it *filters* potentially false dependencies in order to speed up the validation process. Afterwards, it *validates* each dependency by dynamic analysis and it *recovers* any manifest dependency that is potentially missing in the initial and/or filtered graph.

The output of TEDD is a validated TDG, which ensures that any test execution schedule that respects its dependencies will not result in any test failure. In our empirical study on six web test suites, TEDD produced the final TDGs 72% faster than a baseline approach that validates all possible dependencies (57%, on average). Also, the test suites parallelized by TEDD according to the dependencies in the final TDGs achieved a speedup up to 7× (2× on average).

Our paper makes the following contributions:

- The first test dependency detection approach for web tests. Our approach introduces string analysis (SA) to extract an approximated set of test dependencies, and NLP/SA to filter potential false dependencies.
- An algorithm to automatically retrieve all missing dependencies from any given web test dependency graph.
- An implementation of our algorithm in a tool named TEDD, which is available [38].
- An empirical evaluation of TEDD on a benchmark of six open-source web test suites, comprising 196 test cases.

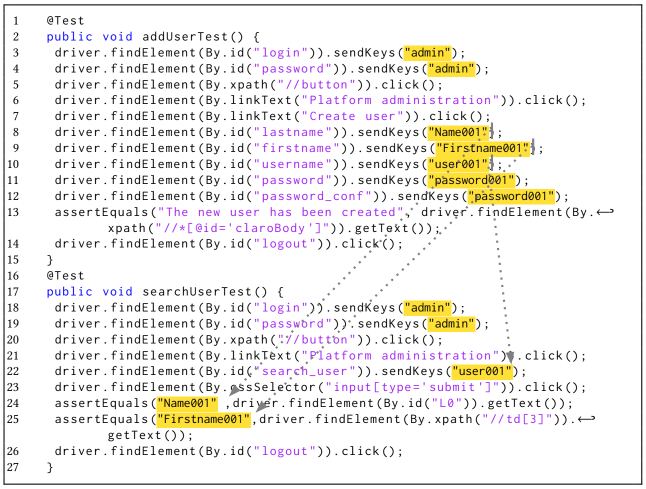
2 BACKGROUND AND MOTIVATION

Ideally, running the tests in a test suite in any order should produce the same outcome [12]. This means tests should deterministically pass or fail *independently* from the order in which they are executed. A test dynamically alters the state of the program under test in order to assert its expected behaviour. In practice, some tests fail to undo their effects on the program's state after their execution, which can pollute any shared state [17] in tests executed subsequently.

In the web domain, testers perform end-to-end (E2E) testing of their applications [4, 11] by creating test cases using test automation tools such as Selenium WebDriver [34]. Such tests consist of (1) actions that simulate an end-user's interactions with the application, and (2) assertions on information retrieved from the web page to verify the expected behaviour. Unlike unit testing, in which tests target specific class methods, web tests simulate E2E user scenarios, and therefore the program state that persists across test case executions might be left polluted, causing test failures if tests are reordered.

Table 1: Test cases for Claroline, numbered according to their test execution order [24].

Test	Name	Description
t_1	addUserTest	The admin creates a new user account.
t_2	searchUserTest	The admin searches for the newly created user.
t_3	loginUserTest	The newly created user logs in to the application.
t_4	addCourseTest	The admin creates a new course.
t_5	searchCourseTest	The admin searches for the newly created course.
t_6	enrolUserTest	The user enrols herself in the course.



```
1    @Test
2    public void addUserTest() {
3     driver.findElement(By.id("login")).sendKeys("admin");
4     driver.findElement(By.id("password")).sendKeys("admin");
5     driver.findElement(By.xpath("//button")).click();
6     driver.findElement(By.linkText("Platform administration")).click();
7     driver.findElement(By.linkText("Create user")).click();
8     driver.findElement(By.id("lastname")).sendKeys("Name001");
9     driver.findElement(By.id("firstname")).sendKeys("Firstname001");
10    driver.findElement(By.id("username")).sendKeys("user001");
11    driver.findElement(By.id("password")).sendKeys("password001");
12    driver.findElement(By.id("password_conf")).sendKeys("password001");
13    assertEquals("The new user has been created", driver.findElement(By.↩
             xpath("//*[@id='claroBody']")).getText());
14    driver.findElement(By.id("logout")).click();
15   }
16   @Test
17   public void searchUserTest() {
18    driver.findElement(By.id("login")).sendKeys("admin");
19    driver.findElement(By.id("password")).sendKeys("admin");
20    driver.findElement(By.xpath("//button")).click();
21    driver.findElement(By.linkText("Platform administration")).click();
22    driver.findElement(By.id("search_user")).sendKeys("user001");
23    driver.findElement(By.cssSelector("input[type='submit']")).click();
24    assertEquals("Name001" ,driver.findElement(By.id("L0")).getText());
25    assertEquals("Firstname001",driver.findElement(By.xpath("//td[3]")).↩
             getText());
26    driver.findElement(By.id("logout")).click();
27   }
```

Figure 1: Two dependent E2E web tests for the Claroline web application. Potential dependencies due to shared input data are highlighted.

Motivating Example. Table 1 lists six E2E Selenium WebDriver tests for the Claroline web application [24], one of the subject test suites used in our evaluation.

Figure 1 shows dependencies between tests t_1 and t_2. The test case addUserTest logs in to the application with the administrator credentials (lines 3–5), it navigates to the create user page (lines 6–7), it creates a new user account having username user001 by filling in and submitting the appropriate form (lines 8–12), and it finally verifies that a message is correctly displayed (line 13).

The execution of addUserTest pollutes the state of the web application, which is used by the subsequent test searchUserTest to search for the same user user001 created by addUserTest (line 22). Thus, the shared input data user001 might reveal a potential dependency between the tests (see highlighted inputs in Figure 1).

To make these two tests independent and avoid polluted program states, a tester, for instance, should (1) delete the user user001 created in addUserTest, to clean the polluted program state, and (2) re-create the same user (or a different one) in searchUserTest.

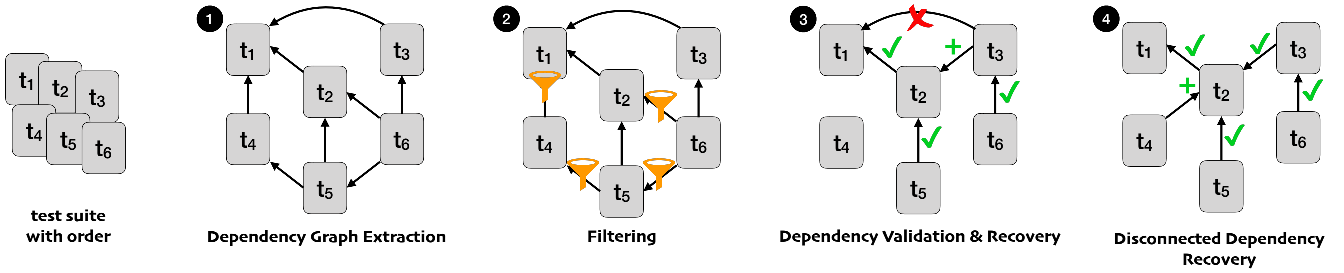

t1
t2
t3
t4
t5
t6
test suite with order
1
Dependency Graph Extraction
2
Filtering
3
Dependency Validation & Recovery
4
Disconnected Dependency Recovery

Figure 2: Our overall approach for web test dependency detection, validation and recovery.

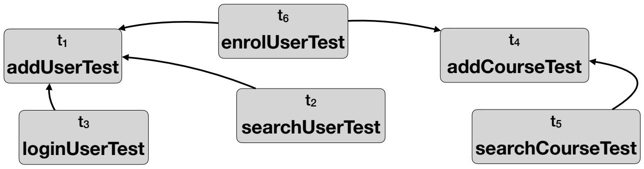


Figure 3: Test dependency graph for the test suite of Table 1. Solid edges represent *manifest* dependencies, namely, dependencies that result in a different outcome if unrespected.

In practice, however, testers re-use states created by preceding tests to avoid test redundancy, higher test maintenance cost and increased test execution time [25]. In doing so, they also enforce pre-defined test execution orders, which in turn inhibit utilizing test optimization techniques such as test prioritization [33].

Test Dependency Graph. Dependencies occurring between tests can be represented in a test dependency graph (*TDG*) [12], a directed acyclic graph in which nodes represent test cases and edges represent dependencies. *TDG* contains an edge from a test t_2 to a test t_1 if t_2 depends on t_1 for its execution (notationally, $t_2 \longrightarrow t_1$).

Figure 3 illustrates the actual test dependency graph (*TDG*) for the test suite of Table 1. For example, *TDG* contains an edge from searchCourseTest to addCourseTest because searchCourseTest requires the execution of addCourseTest to produce the expected result (in other words, addCourseTest *must be executed before* searchCourseTest in order for it to succeed). Multiple test dependencies can also occur. For instance, enrolUserTest depends on both addUserTest and addCourseTest for its correct execution.

In order to be useful, *TDG* should contain all *manifest dependencies*, i.e., dependencies that do cause tests to fail if violated, while retaining the minimum number (or none) of *false dependencies.*

One possible application of a *TDG* that contains only manifest dependencies is test suite parallelization. For instance, if we traverse the graph of Figure 3 and extract the subgraphs reachable from each node with zero in-degree (in our example, there is only addUserTest), we can identify subsets of tests that can be be executed in parallel with the others. In our example, four parallel test suites are possible: $\{ \langle t_1, t_2 \rangle, \langle t_1, t_3 \rangle, \langle t_1, t_4, t_6 \rangle, \langle t_4, t_5 \rangle \}$.

3 APPROACH

The goal of our approach is to automatically detect the occurrence of dependencies among web tests. Detecting dependencies in E2E web tests is particularly challenging due to the stack of programming languages and technologies involved in the construction of modern web applications, e.g., HTML, CSS, JavaScript on the client side; PHP, Java or JavaScript on the server-side, and back-end layers through Restful APIs or databases.

Our insight is that by analyzing the input data used in test cases (highlighted in Figure 1), we can obtain clues about potential test dependencies caused by shared polluted states. For example, the input string user001 used at line 10 in addUserTest is a *write operation* of persistent data. The same input string is used at line 22 in searchUserTest as a *read operation* of persistent data. We conjecture that such *read-after-write* connections on persistent data could indicate potential test dependencies. Given this insight, our approach focuses on read-after-write relationships on persistent data, defined as follows.

Definition 1 (Persistent Read-After-Write (PRAW) Dependency). *Two test cases t_1 and t_2 executed in this order in the original test suite are subject to a PRAW dependency if t_1 performs an operation that writes some information S_i into the persistent state of the web application and t_2 performs an operation that reads S_i from the persistent state of the web application.*

Examples of the write operations in t_1 include creating, updating or deleting a record in a database, or creating a DOM element on the webpage. Examples of read operations in t_2 are reading the same record from the database, or accessing the newly created DOM element on the webpage.

Figure 2 illustrates our overall approach, which requires a web test suite as input, along with a predefined test execution order. Overall, our approach ❶ computes an initial approximated test dependency graph, ❷ filters out potential false dependencies, ❸ dynamically validates all dependencies in the graph while recovering any missing dependencies, and, finally, ❹ handles missing dependencies affecting independent nodes possibly resulting from the previous validation step.

Next, we describe each step of our approach.

3.1 Dependency Graph Extraction

In the first step, from the input test suite, our approach computes an initial test dependency graph representing an approximated set of candidate dependencies. This can be conducted in different ways, as described below.

Algorithm 1: Sub-use string analysis graph extraction

```
    Input  : $T_o$: test suite in its original order $o$, $\mathcal{I}$: set of input-submitting actions
    Output: $TDG$: test dependency graph with candidate dependencies to be validated
1   $TDG \leftarrow \emptyset$
2   $T_a \leftarrow T_o$                                                  ▷ tests to analyze
3   foreach $t$ in $T_o$ do
4       $\mathcal{S} \leftarrow$ getInputValues($t$, $\mathcal{I}$)        ▷ $\mathcal{S}$ is the set of input values submitted by $t$
5       $T_a \leftarrow T_a - \{t\}$
6       foreach $t_f$ in $T_a$ do
7           $\mathcal{U} \leftarrow$ findUsedValues($t_f$) $\cap \mathcal{W}$   ▷ $\mathcal{U}$ is the set of input values used by $t_f$
8           if $\mathcal{U} \neq \emptyset$ then
9               $depToAdd \leftarrow (t_f \xrightarrow{\mathcal{U}} t)$           ▷ candidate manifest dependency
10              $TDG \leftarrow TDG \cup \{ depToAdd \}$
11          end
12      end
13  end
```

3.1.1 Original Order Graph Extraction. A baseline approach consists of connecting all pairwise combinations of tests according to the original order. This results in a directed graph in which every pair of distinct nodes is connected by a unique pair of edges, so as to establish a dependency relation between each test and all the others that are executed before it. If n is the number of test cases in the test suite, the graph contains $\frac{n(n-1)}{2}$ edges (e.g., dependencies).

Since the time to validate a dependency graph increases with the number of dependencies in the graph, heuristics can be used to reduce the size of the graph by removing edges that are less likely to be manifest dependencies. To that end, we propose an approach that leverages a fast static *string analysis* of the input data present in the tests, to construct a smaller initial test dependency graph.

3.1.2 Sub-Use String Analysis Graph Extraction. Algorithm 1 describes our dependency graph extraction based on sub-use-chain relations. A sub-use relation consists of a submission (*sub*) of input data *i* and all the following *uses* that submitted value *i*.

Starting from the first test case according to the original test suite order, Algorithm 1 first retrieves the set $\mathcal{S}$ of input values *submitted* by the test, like values inserted into input fields by input-submitting actions such as the `sendKeys` methods (line 4).

Second, the algorithm considers each test case t_f following t and searches for any input value in the set $\mathcal{S}$, which is *used* in any statement of the current test case t_f (line 7), and adds them to the set of *used* values $\mathcal{U}$. If at least one string value is found (i.e., a *sub-use* chain), a candidate PRAW dependency between t and t_f is created by adding the edge $t_f \longrightarrow t$, labelled with each retrieved string value (line 9), to the test dependency graph (line 10).

The internal loop of Algorithm 1 (lines 6–12) searches for the input values indistinctly in any test action because in web tests there is no clear distinction between read and write statements. For instance, in Figure 1, the `sendKeys` action at line 10 is used to *write* persistent information into the application (e.g., in a database). However, the same `sendKeys` action, at line 22, identifies an action with a *read* connotation, because the string value `user001` is used to search for a specific persistent information in the web application.

Let us consider the source code of the two test cases in Figure 1, namely `addUserTest` and `searchUserTest`. From the first test `addUserTest`, the algorithm extracts the set $\mathcal{S}$ = {`admin`, `Name001`, `Firstname001`, `user001`, `password001`} because `sendKeys` is the only input-submitting action. Then, the string values in $\mathcal{S}$ are looked up in the subsequent test `searchUserTest`, producing the set of used values $\mathcal{U}$ = {`admin`, `Name001`, `Firstname001`, `user001`}. Being $\mathcal{U}$ not empty, the algorithm creates a candidate test dependency between `searchUserTest` and `addUserTest`.

3.2 Filtering

The second step of our approach applies a filtering process to remove potential false PRAW dependencies. The filtering is performed to speed up the subsequent validation step, which requires in-browser test execution, and therefore can be computationally expensive for graphs with numerous candidate test dependencies. Finding an effective filtering technique is, however, challenging. A *loose* filter might remove a few false dependencies, whereas a *strict* filter might mistakenly remove manifest dependencies, which would need to be recovered at a later stage.

In this work, we propose two novel test dependency filtering techniques based on (1) dependency-free values, and (2) Natural Language Processing (NLP).

3.2.1 Dependency-free String Value Filtering. We analyze the frequency of string input values used in the test suite to filter potential false PRAW dependencies.

Let us consider the test dependency graph depicted in Figure 4, obtained by applying our sub-use string analysis graph extraction (Section 3.1.2) to the motivating example test suite (Section 2).

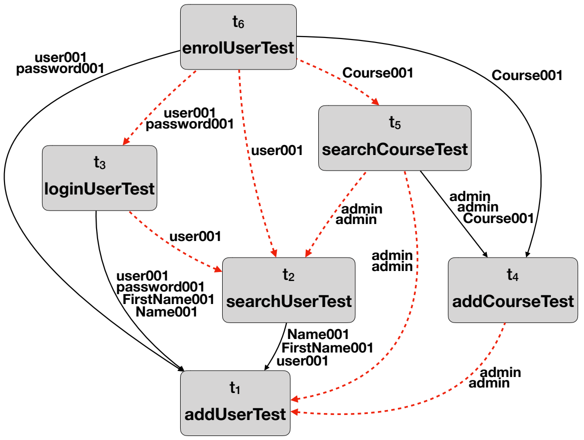


Figure 4: Dependency-free string-based PRAW filtering. Solid black edges represent manifest dependencies whereas dashed red edges represent false dependencies.

The set of dependencies $\{ t_5 \longrightarrow t_2, t_5 \longrightarrow t_1, t_4 \longrightarrow t_1 \}$ represents instances of *false* candidate PRAW dependencies. The edges between these test cases are only due to the same login input data used by the tests—i.e., `admin`—a default user created during the installation, for which no test case must be executed to create it.

Existing techniques [12, 42] refer to such cases as *dependency-free values*, i.e., if the test dependency graph includes dependencies that are shared across multiple (or all) test cases, these likely-false dependencies could be filtered out.

However, in principle, these assumptions might not hold in all cases, as occurrence frequency alone is not conclusive for safe filtering. Our dependency-free string value filtering computes a ranked list of frequently occurring strings and asks the developer to either confirm or discard them (if a string value occurs in all test cases, the corresponding dependency is automatically filtered).

In our example of Figure 4, our approach computes the frequencies of all strings, presents it to the the tester, who, for instance, may decide to filter the dependencies due to the `admin` string, hence removing $t_5 \longrightarrow t_2$, $t_5 \longrightarrow t_1$, $t_4 \longrightarrow t_2$.

3.2.2 NLP-based Filtering. Developers often use descriptive patterns for test case names, which summarize the operations performed by each test. Giving a descriptive name to a test case has several advantages such as enhanced *readability* (i.e., it becomes easier to understand what behaviour is being tested) and *debugging* (i.e., when a test case fails, it is easier to identify the broken functionality). For instance, Google recommends test naming conventions [15, 16] in which *unit tests* need to be named with the method being tested (a verb or a verb phrase, e.g., `pop`) and the application state in which the specific method is tested (e.g., `EmptyStack`). For *behaviour-based* tests such as E2E tests [16], the guidelines propose a naming convention that includes the test scenario (e.g., `invalidLogin`) and the expected outcome (e.g., `lockOutUser`).

Therefore, our second filtering mechanism consists of using Natural Language Processing (NLP) to analyze test case names and classify them into two classes, namely, *read* or *write*. Then, based on such classification, non-PRAW dependencies, such as a "read" test being dependent on another "read" test, are discarded from the test dependency graph.

Our approach uses a *Part-Of-Speech tagger* (POS) to classify each token (i.e., word) in a tokenized test name as noun, verb, adjective, or adverb. In particular, our approach uses the verb from the test case name as the part of speech that conveys the nature of the test operation, and uses it to classify each test into *read* or *write* classes. Our approach relies on two groups of standardized R/W verbs, namely CRUD operations—*Create*, *Read*, *Update* and *Delete*) [10]—in which the *Read* operation is pre-classified as read whereas the other three are pre-classified as write.

Our approach uses POS to extract the first verb from each test name and then computes the semantic similarity [32] (specifically, the *WUP* metrics [41]) between the extracted verb and each verb in the pre-classified read/write groups. The similarity score quantifies how much two concepts are alike, based on information contained in the *is–a* hierarchy of *WordNet* [30]. After computing all similarity scores, our approach classifies the extracted verb to the group having the maximum similarity score. In case of ties (e.g., the verb has the same similarity score for both the read and write classes), or in case no verbs are found, our approach does not perform any assignment and the dependency is not filtered. Our classification of read/write verbs achieved a precision of 80% and a recall of 94% on our experimental subjects.

In this work, we propose and evaluate three NLP configurations.

Verb only (NLP Verb). Our first NLP filtering configuration considers only the *verb of the test case name*. Given a dependency $t_y \longrightarrow t_x$, our approach extracts the verb from both t_y and t_x, and classifies them either as read or write. Then, it filters (1) the read-after-read (RaR) dependencies, in which both t_y and t_x have verbs classified as read, and (2) the write-after-read (WaR) dependencies, where t_y has a write-classified verb whereas t_x has a read-classified verb. All other types of dependencies, such as read-after-write (RaW) and WaW (write operations in web applications often also involve reading existing data), are retained.

The dependency `searchCourseTest` ⟶ `searchUserTest` (Figure 4) is filtered because it is classified as RaR, with `search` being the read-classified verb. Conversely, the edge `searchUserTest` ⟶ `addUserTest` is retained since it is classified as RaW, being `search` and `add` the read/write verbs, respectively. The word `Test` is considered a *stop word* and removed before the NLP analysis starts.

Verb and direct object (NLP Dobj). The second configuration considers the *direct object* the verb refers to. Given a set of test cases, our approach uses a dependency parser to analyze the grammatical structure of a sentence, extract the direct object from each test name, and construct a set of "dobject" dependencies.

RaR and WaR dependencies are filtered as described in the previous NLP Verb case. Differently, RaW and WaW dependencies are filtered only if the direct objects of two verbs appearing in two tests t_y and t_x are different. The intuition is that the two tests may perform actions on different persistent entities of the web application, if the involved direct objects in the test names are different. For example, in Figure 4, the RaW dependency `searchCourseTest` ⟶ `addUserTest` is filtered because the two involved direct objects, `Course` and `User`, are different.

Verb and nouns (NLP NOUN). Our third configuration takes into account all entities of type *noun* contained in the test names. When the test name includes multiple, different entities, analyzing only the direct object may not be enough to make a safe choice. For instance, in our subject Claroline, the analysis of the direct object would erroneously filter the manifest dependency `addCourseEventTest` ⟶ `addCourseTest`, because it is a WaW and the two direct objects `Event` and `Course` are different. However, there is an implicit relation between the direct object `Event` and the `Course` object it refers to. Thus, the dependency with `addCourseTest` should be retained.

Again, RaR and WaR dependencies are filtered as described in the NLP Verb case. Here, RaW and WaW dependencies are filtered only if the two tests involved in a dependency have no noun in common. As such, the manifest dependency `addCourseEventTest` ⟶ `addCourseTest` would not be filtered in this configuration, because of the shared name `Course`.

3.3 Dependency Validation and Recovery

Given a test dependency graph *TDG*, the overall dynamic dependency validation procedure works according to the iterative process proposed by Gambi et al. [12]. The approach executes the tests according to the original order to store the expected outcome. Next, it selects a target dependency according to a source-first strategy in which tests that are executed later in the original test suite are selected *first* (i.e., $t_3 \longrightarrow t_2$ would be selected before $t_2 \longrightarrow t_1$).

To validate the target dependency, tests are executed out of order, i.e., a test schedule in which the target dependency is *inverted* is computed and executed. If the result of the test execution differs from the expected outcome, the target dependency is marked as *manifest*, because the failure was due to the inversion. Otherwise, the target dependency is removed from *TDG*. The process iterates until all dependencies are either removed or marked as manifest.

Algorithm 2: Recovery algorithm

```
Input   : $T_o$: test suite in its original order $o$
          $TDG$: test dependency graph
          targetDep: dependency selected for validation
          expResults: results of executing $T_o$
          execResults: results of a test schedule in which targetDep is inverted
Output  : $TDG$: updated test dependency graph with missing dependencies recovered
Require : expResults ≠ execResults, i.e., targetDep is manifest
1  schedule ← computeTestScheduleWithNoInversion($TDG$, targetDep)
2  execResults ← executeTestSchedule(schedule)
3  failedTest ← getFirstFailedTest(expResults, execResults)
4  if failedTest ≠ null then
5      /* Failure due to a missing dependency; get all tests before failedTest. */
6      depCandidates ← getDepCandidates($T_o$, schedule)
7      foreach depCandidate ∈ depCandidates do
8          depToAdd ← ⟨ failedTest → depCandidate ⟩
9          $TDG$ ← $TDG$ ∪ {depToAdd}
10     end
11 end
```

The dynamic validation procedure described above works correctly under the assumption that the initial TDG contains all manifest dependencies (as the original order graph Section 3.1.1). In our approach, the filtering techniques applied in the previous step may be not conservative. Therefore, differently from existing techniques [12], our approach features a dynamic dependency recovery mechanism that retrieves all *missing* dependencies. To the best of our knowledge, this is the first dependency validation algorithm that also includes dynamic dependency recovery.

Recovering Missing Dependencies. Algorithm 2 takes a partially-validated TDG. For each failing test schedule in which a target dependency is inverted, it checks whether the failure is due to a missing dependency in the dependency graph.

More specifically, Algorithm 2 takes the target dependency and computes a schedule in which the target dependency *is not* inverted (line 1). If the execution of such schedule complies with the expected outcome, our approach considers the test failure due to the dependency inversion and marks the dependency as a manifest. On the contrary, if one or more tests fail also in the schedule without inversion (line 4), our approach assumes that one or more dependencies are missing and need to be recovered.

To do so, the algorithm takes the first failing test and retrieves the preceding test cases that were not executed in the schedule (line 6). Those tests are all candidate manifest dependencies for the failed test. The algorithm connects the failed test case with each such preceding test and adds those dependencies to the graph (lines 8–9). The graph obtained this way contains all newly added candidate manifest dependencies that still need to be validated.

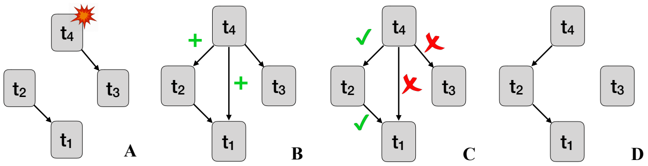


Figure 5: Recovery of missing dependencies. (A) $t_4 \longrightarrow t_3$ is selected, t_4 fails because $t_4 \longrightarrow t_2$ is missing. (B) recovery procedure adds candidate dependencies. (C) dependency validation. (D) final TDG.

Let us take as example Figure 5.A, in which t_4 has a *missing manifest dependency* on t_2, and t_3 does not modify the application state. According to the source-first strategy, the validation selects the dependency $t_4 \longrightarrow t_3$. The schedule computed for such dependencies is $\langle t_4 \rangle$, in which t_4 fails because t_2 is not executed. Then, our algorithm starts retrieving the missing dependency by computing a schedule in which $t_4 \longrightarrow t_3$ is not inverted, $\langle t_3, t_4 \rangle$, in which t_4 fails again for the same reason. The recovery procedure concludes that there is at least one missing dependency, and connects t_4 with both t_1 and t_2 (Figure 5.B), i.e., the only candidate manifest dependencies.

In Figure 5.C the dependency $t_4 \longrightarrow t_3$ is selected again. This time, the computed schedule is $\langle t_1, t_2, t_4 \rangle$, in which none of the tests fail. Therefore, the dependency is marked as false and removed. The next selected dependency is $t_4 \longrightarrow t_2$, for which the schedule $\langle t_1, t_4 \rangle$ is computed. The test t_4 fails because t_2 is not executed. To check if the failure is due to a missing dependency, our algorithm computes the test schedule $\langle t_1, t_2, t_4 \rangle$, in which none of the tests fail. Our algorithm concludes that $t_4 \longrightarrow t_2$ is a manifest dependency and *recovers* it. The validation iterates over the other dependencies in the same way and outputs the final TDG (Figure 5.D), where the initially missing dependency has been recovered.

3.4 Disconnected Dependency Recovery

The previous validation step ❸ can produce disconnected components in the TDG. Missing dependencies involving tests in disconnected components require a separate treatment. Two cases can occur (1) tests with no outgoing edges (zero out-degree),[1] and (2) *isolated* tests, i.e., tests having neither incoming nor outgoing edges (zero in- and out-degree).

The former case occurs when a false dependency, removed during the validation, *shadows* a missing dependency. In such cases, disconnected components of TDG, including potentially missing dependencies, are created as a result of the validation.

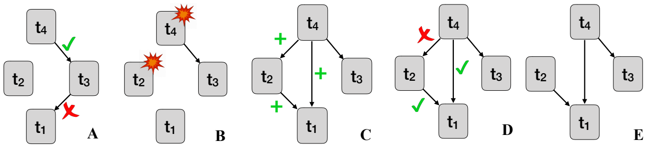


Figure 6: Disconnected dependency recovery. (A) dependencies are validated; $t_3 \longrightarrow t_1$ shadows the missing dependency $t_4 \longrightarrow t_1$. (B) t_4 and t_2 fail because of missing dependencies. (C) recovery procedure adds candidate dependencies. (D) dependencies are validated. (E) final TDG.

Figure 6.A illustrates an example: the manifest dependency $t_4 \longrightarrow t_1$ is missing in the initial dependency graph. Let us suppose that t_3 does not change the state of the application when executed, and therefore, its execution does not influence the execution of any other successive test in the original order. The algorithm selects first the dependency $t_4 \longrightarrow t_3$, it produces the schedule $\langle t_4 \rangle$, in which the test fails since t_1 is not executed. Our approach checks if the failure is due to a missing dependency. When the dependency is *not inverted*, the computed schedule is $\langle t_1, t_3, t_4 \rangle$, in which none

[1] First test t_1 excluded

of the tests fail. Hence, our algorithm concludes that $t_4 \longrightarrow t_3$ is a manifest dependency and no dependency recovery takes place. In the next step, the algorithm validates the dependency $t_3 \longrightarrow t_1$, which is removed because t_3 can execute successfully without t_1. The dependency graph produced by dependency validation algorithm is illustrated in Figure 6.B. In the isolated subgraph $t_4 \longrightarrow t_3$ the schedule $\langle t_3, t_4 \rangle$ results in a failure of t_4. Indeed, the dependency $t_4 \longrightarrow t_1$ was not captured by the recovery algorithm because *the false dependency* $t_3 \longrightarrow t_1$ *shadowed the absence of the manifest dependency* $t_4 \longrightarrow t_1$.

Figure 6.A also illustrates how our approach handles isolated tests. In this example, t_2 is an isolated node. Let us suppose that t_2 has a manifest dependency on t_1 ($t_2 \longrightarrow t_1$), which is missing in the initial *TDG* because it is either not captured, or because it is wrongly filtered out in the second step of our approach. Therefore, the validation step would produce the *TDG* shown in Figure 6.B, in which there is no chance to check whether t_2 executes successfully in isolation. In fact, t_2 is not part of any test schedule that can be generated from *TDG*, regardless of any possible dependency inversion. For this reason, a further recovery step is required once the validation is completed.

Algorithm 3: Disconnected dependency recovery algorithm

```
   Input  : $T_o$: test suite in its original order $o$
            $TDG$: test dependency graph
   Output : $TDG$: updated test dependency graph with missing dependencies recovered
1  $expResults \leftarrow$ EXECUTETESTSUITE($T_o$)
2  /* Get isolated nodes and nodes with no outgoing edges. */
3  $disconnectedTests \leftarrow$ GETDISCONNECTEDTESTS($TDG$)
4  foreach $disconnectedTest$ in $disconnectedTests$ do
5      $execResults \leftarrow$ EXECUTETESTINISOLATION($disconnectedTest$)
6      $failedTest \leftarrow$ GETFAILEDTEST($expResults$, $execResults$)
7      if $failedTest \neq null$ then
8          $TDG \leftarrow$ CONNECTWITHPRECEDINGTESTS($failedTest$, $TDG$, $T_o$)
9      else if ISNOTISOLATED($disconnectedTest$) then
10         /* Out-degree = 0; in-degree > 0. */
11         $schedules \leftarrow$ COMPUTESCHEDULES($disconnectedTest$, $TDG$)
12         foreach $schedule \in schedules$ do
13             $execResults \leftarrow$ EXEC(schedule)
14             $failedTest \leftarrow$ GETFAILEDTEST($expResults$, $execResults$)
15             if $failedTest \neq null$ then
16                 $TDG \leftarrow$ CONNECTWITHPRECEDINGTESTS($failedTest$, $TDG$, $T_o$)
17             end
18         end
19     end
20 end
```

Algorithm 3 handles the recovery of missing dependencies within disconnected components. The algorithm retrieves all isolated nodes and zero out-degree nodes (line 3) and executes each of them in isolation (line 5). For each failing test, the algorithm connects it with all preceding tests according to the initial test suite order (line 8). Otherwise, if a test is not isolated and executes successfully (line 9), the algorithm takes all schedules that contain that test and execute them (lines 12–18). If a test in those schedules fails, the algorithm connects it with all the preceding ones (line 16).

Finally, for each dependency found and added to *TDG* during the disconnected dependency recovery step, the dependency validation procedure must be re-executed.

Given the graph in Figure 6.B, Algorithm 3 executes t_2 in isolation, which fails, thus the dependency $t_2 \longrightarrow t_1$ is added. Moreover, in Figure 6.B, there is only one schedule that involves t_3, namely $\langle t_3, t_4 \rangle$. In this schedule t_4 fails, hence our approach adds the dependencies $t_4 \longrightarrow t_2$ and $t_4 \longrightarrow t_1$ (Figure 6.C). Next, the added dependencies are validated (Figure 6.D), and the final graph is produced, where all initially missing manifest dependencies have been successfully recovered (Figure 6.E).

To conclude, our validation and recovery algorithms makes sure that (1) newly added dependencies are themselves validated, (2) false dependencies are removed in the final *TDG*. Indeed, a node in the final *TDG* can be either (i) connected (i.e., in-degree > 0 and out-degree > 0), (ii) without outgoing edges (i.e., in-degree > 0 and out-degree = 0) or (iii) isolated (i.e., in-degree = out-degree = 0).

3.5 Implementation

We implemented our approach in a Java tool called TEDD (**Te**st **D**ependency **D**etector), which is available [38]. The tool supports Selenium WebDriver web test suites written in Java. TEDD expects as input the path to a test suite and performs the string analysis by parsing the source code of the tests by using *Spoon* (version 6.0.0) [31]. Our NLP module adopts algorithms available in the open-source library *CoreNLP* (version 3.9.2) [9]. The output of TEDD is a list of manifest dependencies extracted from the final validated TDG.

4 EMPIRICAL EVALUATION

We consider the following research questions:

RQ$_1$ (effectiveness): How effective is TEDD at filtering false dependencies without missing dependencies to be recovered?

RQ$_2$ (performance): What is the overhead of running TEDD? What is the runtime saving achieved by TEDD with respect to validating complete test dependency graphs?

RQ$_3$ (parallel test execution): What is the execution time speed-up of the test suites parallelized from the test dependency graphs computed by TEDD?

4.1 Subject Systems

We selected six open-source web applications used in previous web testing research [24]. Each subject comes with one JUnit 4 test suite containing between 22-41 Selenium test cases; a JUnit test runner class specifies the tests order provided by the developer. Table 2 lists our subject systems, including their names, version, size in terms of lines of code, number of test cases, and the total number of lines of test code counted with `cloc` [8].

Table 2: Subject systems and their test suites

	WEB APP		TEST SUITES	
	Version	LOC	#	LOC (Avg/Tot)
Claroline	`1.11.10`	352,537	40	46/1,822
AddressBook	`8.0.0`	16,298	27	49/1,325
PPMA	`0.6.0`	575,976	23	54/1,232
Collabtive	`3.1`	264,642	40	48/1,935
MRBS	`1.4.9`	34,486	22	51/1,114
MantisBT	`1.1.8`	141,607	41	43/1,748
Total		866,995	196	47/9,176

4.2 Procedure and Metrics

4.2.1 Procedure. We manually fixed any flakiness of the test cases of the subject test suites by adding delays where appropriate and we executed each test suite 30 times to ensure that identical outcomes are obtained across all executions.

To form a *baseline* for comparison, we applied dependency validation to the dependency graph obtained from the original order of each test suite (Section 3.1.1). Essentially, our baseline approach represents the execution of Pradet's [12] heuristics on the *TDG*s obtained from the original test suites.

For each test suite, we ran different configurations of TEDD, by combining each admissible combination of graph extraction and filtering technique. The first evaluated configuration is *String Analysis* (SA), in which the dependency graph is obtained through sub-use chain extraction (Section 3.1.2) and filtered from the dependency-free values (Section 3.2.1). Then, we evaluated three configurations in which we applied the three proposed NLP filters (NLP-Verb, NLP-Dobj, NLP-Noun) both to the graph from the original order as well as to the graph obtained with SA.

Finally, given the validated dependency graph obtained in each configuration, we generated all possible parallel test *schedules* automatically, by traversing the final validated TDG from dependents to dependees, until all tests are included. Each parallelized test suite was executed sequentially.

4.2.2 Metrics. To assess *effectiveness* (RQ_1), for each configuration we measured the number of *false dependencies* removed by TEDD as well as the number of manifest dependencies that are *missing* and need to be *recovered*. The number of false dependencies is obtained by subtracting the number of manifest dependencies retrieved at the end of the recovery step from the total number of dependencies in the initial graph.

We evaluated *performance* (RQ_2) by comparing the execution time (in minutes) of each configuration of TEDD with respect to the baseline approach.

Concerning *parallelization* (RQ_3), we measured the speed-up factor of the parallelizable test suites with respect to the original test suite running time. We considered two speed-up scenarios. (1) *average case*, in which we measured the ratio between the original test suite running time and the average running time of the parallelizable test suites, and (2) *worst case*, in which we measured the speed-up ratio between the original test suite running time and the parallelizable test suite having the highest runtime.

4.3 Results

Effectiveness (RQ_1). For each configuration of TEDD, Table 3 (Effectiveness) shows the number of *extracted* dependencies starting from the initial test suite (Figure 2, step ❶) and the number of *filtered* dependencies (Figure 2, step ❷). It also reports information about the validation and recovery steps, specifically the number of *false* dependencies detected, the number of dependencies *recovered* and those recovered from the disconnected components. The final number (Column 8) shows the number of dependencies in the final *TDG*s, all of which are *manifest* PRAW dependencies.

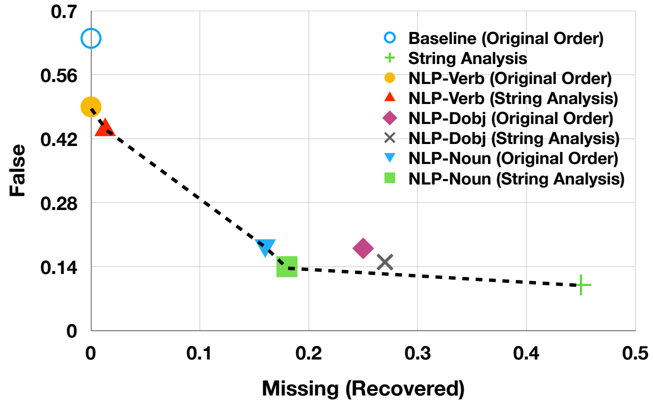


Figure 7: Pareto front

Across all apps, the baseline approach validated on average 536 dependencies, of which 504 were deemed as false, and 32 as manifest. The most conservative among TEDD's configurations is NLP-Verb (Original Order), which validated overall 416 dependencies, of which 384 were false (24% less than the baseline) and detecting 32 manifest dependencies without filtering/recovering any. The least conservative configuration of TEDD is NLP-Noun (String Analysis) which retained only 143 dependencies on average from the initial graphs, of which 110 were detected as false, five dependencies had to be recovered, leading to the final number of 33 manifest dependencies. Overall, the number of missing dependencies due to filtering and recovered in steps ❸ ❹ is very low (1% of the initial number of dependencies).

TEDD does not ensure having minimal test dependency graphs. Therefore, the number of manifest dependencies retrieved by each configurations is slightly different, between 32 and 34 (Column *Total PRAW*). However, these differences do not affect the executability of the schedules that respect the dependencies (see results for RQ_3).

Figure 7 shows the Pareto front plotting the ratio between false and missing dependencies, for each configuration. Each point represents the average ⟨*missing*, *false*⟩ values across all subjects, normalized over the respective maximum values. This essentially shows the tradeoff between the false dependencies remaining after the filtering step and the missing dependencies to be recovered.

From the analysis of the Pareto front, we can see that the non-dominated configurations are those based on NLP-Verb, NLP-Noun and String Analysis (SA). The baseline approach (Baseline) has the highest number of false dependencies (536 on average) and no missing dependencies. On the contrary, SA filters many dependencies (393 on average) but has the highest number of missing/recovered dependencies (11 on average). Interestingly, NLP-Verb (Original Order) does not miss any manifest dependency but has more false dependencies compared to the other NLP-based configurations. Configurations NLP-Dobj (both SA and Original Order) and NLP-Noun (both SA and Original Order) are comparable regarding the number of false dependencies remaining after filtering. However, NLP-Noun (SA and Original Order) needs to recover substantially less manifest dependencies. Indeed, NLP-Noun dominates NLP-Dobj, while both NLP-Noun (SA and Original Order) configurations

Table 3: Effectiveness (RQ1), Performance (RQ2) and Parallelization (RQ3) average results across all subject test suites.

	Effectiveness								Performance						Parallelization		
					Manifest Deps.						Validation					Speed-up (%)	
	Extracted	Filtered	To Validate	False	Validated	Recovered	Recovered (Disc.)	Total PRAW	Extraction	Filtering	Val. and Recovery	Recovery (Disc.)	Total	Saving (%)	Schedules (#)	Worst-case	Average
Baseline (Original Order)	536	-	536	504	32	0	0	32	$0.00^{\dagger}$	-	424.7^{*}	-	424.70	-	30	2.2×	7.1×
String Analysis	494	393	101	69	21	10	1	32	0.10	0.00	162.02	5.21	167.33	61%	30	2.4×	7.1×
NLP-Verb (Original Order)	535	119	416	384	32	0	0	32	$0.00^{\dagger}$	0.04	307.20	1.27	308.51	27%	30	2.2×	7.1×
NLP-Verb (String Analysis)	494	113	381	348	32	1	0	33	0.09	0.04	281.10	1.28	282.50	33%	30	2.2×	7.1×
NLP-Dobj (Original Order)	536	362	174	140	27	6	1	34	$0.00^{\dagger}$	0.24	134.90	3.46	138.60	67%	29	2.1×	6.7×
NLP-Dobj (String Analysis)	494	343	151	117	27	5	2	34	0.09	0.23	129.20	9.10	138.62	67%	29	2.1×	6.7×
NLP-Noun (Original Order)	536	364	172	140	28	3	1	32	$0.00^{\dagger}$	0.05	123.30	2.52	125.87	70%	29	2.1×	6.7×
NLP-Noun (String Analysis)	494	351	143	110	28	4	1	33	0.08	0.04	116.10	4.08	120.30	72%	29	2.1×	6.8×

* *only validation, no within-recovery.* † *execution time < 0.01 minutes (0.6 seconds).*

are on the non-dominated front, being both optimally placed in the lower-left quadrant of the Pareto plot.

Performance (RQ$_2$). Table 3 (Performance) reports the average runtime, in minutes, of each step of TEDD across all configurations. The most expensive step of TEDD is validation, especially for what concerns validating the connected part of the graph (Column 12), whereas dependency graph extraction and filtering (Columns 10 and 11) have negligible costs (under one minute on average in all cases). The cost of disconnected components recovery (Column 13) is generally low, ranging from nearly one minute for NLP-Verb to maximum nine minutes for NLP-Dobj (3.8 minutes on average).

The slowest configuration of TEDD is NLP-Verb (Original Order) which is 27% faster on average (almost 2 hours less) than the baseline approach. The fastest configuration of TEDD is NLP-Noun (SA) which is 72% faster on average (5 hours less) than the baseline approach. This result confirms the Pareto front analysis, showing that NPL-Noun (SA) is the most effective configuration of TEDD.

The table reports also the percentage decrease of each configuration with respect to the baseline, which took approximately 425 minutes on average (≈7 hours). Overall, all SA- or NLP-based configurations of TEDD are significantly faster.

Parallel Test Execution (RQ$_3$). Column 15 (*schedules*) reports the average number of test schedules obtained from the final *TDG*s. Isolated nodes in the dependency graphs are counted as (single-test) schedules. Columns 16 and 17 report the relative speed-up of the parallelizable test suites considering the longest test execution schedule (worst-case) and the average case.

First, in our experiments, no test failures occurred in any of the parallelizable test suite produced by any configuration of TEDD. Essentially, this testifies that the dependency validation and recovery algorithm does not miss any manifest dependency.

Overall, all techniques achieve similar speed-up scores, around 2× in the worst case and 7× on average. This is expected since the final TDGs are similar across configurations (see total number of manifest dependencies in Table 3). However, results differ across applications. Table 4 presents the results for the NLP-Noun (SA) configuration. Column *runtime original* reports the execution time of the original test suite in seconds. *Collabtive* has the slowest test suite (almost 5 minutes) whereas *Addressbook* has the fastest (40 seconds). The highest speed-up in the worst-case occurs for *Claroline*, where the longest test execution schedule test suite is 2.7× faster. *MantisBT* exhibits the highest speed-up in the average case (12.3×), but the lowest speed-up in the worst-case (1.4×) due to a single slow-executing schedule (133 s) with respect to the average runtime (15 s). The lowest speed-up in the average case occurs for *MRBS*, but it remains still high (4.2×).

Table 4: Parallelization results for NLP-Noun (SA)

			Worst-case		Average	
	runtime original (min)	schedules (#)	runtime (min)	speed-up (%)	runtime (min)	speed-up (%)
Claroline	75.1	36	29.0	2.7×	9.4	8.3×
Addressbook	40.2	24	20.1	2.0×	8.8	4.5×
PPMA	51.7	22	21.1	2.4×	8.9	5.8×
Collabtive	297.7	37	133.1	2.3×	53.2	5.7×
MRBS	56.9	20	28.7	2.0×	13.8	4.2×
MantisBT	184.5	37	133.8	1.4×	15.1	12.3×
Total	706.1	176	365.9	$2.1\times^{*}$	109.2	$6.9\times^{*}$

* *average*

4.4 Threats to Validity

Using a limited number of test suites in our evaluation poses an *external validity* threat. Although more subject test suites are needed to fully assess the generalizability of our results, we have chosen six subject apps used in previous web testing research, pertaining to different domains, for which test suites were developed by a human web tester. Threats to *internal validity* come from confounding factors of our experiments, such as test flakiness. To cope with possible flakiness of the test cases, we manually fixed any flaky test by adding delays where appropriate and we ran each test suite 30 times to ensure having identical results on all executions. With respect to reproducibility of our results, the source code of TEDD and the *Docker* containers for all subject systems are available online [38], making the evaluation repeatable and our results reproducible.

5 DISCUSSION

Automation and Effectiveness. Our results confirm that (1) E2E web tests entail test dependencies, (2) such dependencies can be identified by considering PRAW connections between test cases, and (3) TEDD can successfully detect all PRAW test dependencies necessary for independent test case execution. All proposed filtering techniques proved both very fast and effective at reducing the size of the initial graph, without filtering many manifest dependencies.

Performance and Overhead. All configurations of TEDD achieve substantial improvements with respect to validating the graph extracted from the original ordering, whose validation cost is quadratic on the number of test cases. During software evolution, when the test suite is modified, our analysis does not need to be re-executed from scratch, as TEDD can harness the dependency information given by a previously validated *TDG*. Validation and recovery, on the other hand, must be carried out on the entire newly produced *TDG*, even though the validation cost is expected to be low, if partial and incremental changes of the test suites are made.

Test Smells. Our test dependency graph can also be utilized for other test analysis activities such as detecting *poorly designed* or *obsolete* tests (i.e., test smells [40]). For instance, in *PPMA*, the test `checkEntryTagsRemoved` executes after `addEntryTags` and `removeEntryTags` tests. By analyzing the *TDG* produced by TEDD for this test suite, we noticed that `checkEntryTagsRemoved` executes properly also when no tags have been created yet (i.e., `checkEntryTagsRemoved` is *isolated* in the *TDG*). This means that the test `checkEntryTagsRemoved` is obsolete because subsumed by the previous `removeEntryTags` test. Therefore, it can be safely removed with no impact on the functional coverage or assertion coverage of the test suite.

Limitations. TEDD depends on the information available in the test source code, used to identify potential test dependencies. As such, the effectiveness of our NLP-based filtering may be undermined if test case names are not descriptive, as in the case of many automatically generated test suites (e.g., `test1`, `test2`). In such cases, testers can rely on the string analysis configuration of TEDD (SA), which also proved effective in our study. Second, our tool does not provide information about the *root cause* of the dependencies, i.e., what part of the program state is polluted by which test. Lastly, TEDD does not support the analysis of flaky test suites.

6 RELATED WORK

Test Dependency Detection. Different techniques have been proposed recently to detect dependencies in unit tests (which is related to the overall problem of test flakiness [3, 19, 20, 22, 29, 35]), none of which focuses on web tests. Zhang et al. [42] developed DTDetector, which detects manifest dependencies in JUnit tests with a dependency-aware k-bounded algorithm. They showed that a small value for k (e.g., $k = 1$ and $k = 2$) finds most realistic dependent tests. The tool VMVM [1] uses test virtualization to isolate unit tests of a JUnit suite, by resetting the static state of the application to its default before each test execution. Our tool TEDD, differently from the previous works, targets the web domain by focusing on dependencies due to read-after-write operations performed on persistent data. ElectricTest [2] utilizes dynamic data-flow analysis to identify all conflicting write and read operations over static Java objects. On the contrary, in E2E web tests, the semantics of read and write operations is implicit and mediated by multiple layers of indirection such as client-side DOM, server-side application state, database entries, and remote service calls. TEDD leverages heuristics based on sub-use-chain relations and NLP to discover potential test dependencies without the need for complex data-flow analysis.

Pradet [12] detects test dependencies in Java unit tests focusing on manifest dependencies that can be traced back to data dependencies. TEDD adopts a similar approach as Pradet to validate the test dependencies, i.e., validating a single dependency at a time by inverting the dependency and linearizing the graph.

The main differences between Pradet and our approach are (1) the capability of TEDD to handle incomplete dependency graphs; Pradet makes the assumption that the initial dependency graph contains all manifest dependencies computed through static analysis of all (read/write) accesses to global Java variables. Due to this assumption, (2) Pradet only removes false dependencies and it does not recover any missing dependencies. TEDD, on the other hand, features a novel recovery algorithm to detect and validate all potentially missing manifest dependencies, which makes it suitable for web E2E test suites, in which applying thorough data-flow analysis is neither feasible nor straightforward, due to the heterogeneity of technologies and languages used in modern web applications. Lastly, (3) TEDD introduces novel extraction and filtering heuristics, which extend the applicability of Pradet beyond read/write operations performed on static fields of Java classes.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we introduce a novel notion of persistent read-after-write (PRAW) dependencies, which extends the standard notion of RAW dependencies for the web domain. Then, we propose a test dependency technique for E2E web test cases based on string analysis and NLP implemented in a tool called TEDD. TEDD achieves an optimal trade off between false dependencies to be removed and missing dependencies to be recovered in six web test suites. In our future work, we plan to apply NLP on the DOM and on page object-based test suites [36], as well as to devise ways to produce a *minimal TDG*, i.e., a *TDG* having the minimum number of dependencies.

REFERENCES

[1] Jonathan Bell and Gail Kaiser. 2014. Unit Test Virtualization with VMVM. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 550–561. DOI: http://dx.doi.org/10.1145/2568225.2568248

[2] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient Dependency Detection for Safe Java Test Acceleration. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 770–781. DOI: http://dx.doi.org/10.1145/2786805.2786823

[3] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 433–444. DOI: http://dx.doi.org/10.1145/3180155.3180164

[4] Stefan Berner, Roland Weber, and Rudolf K. Keller. 2005. Observations and lessons learned from automated testing. In *Proceedings of 27th International Conference on Software Engineering (ICSE 2005)*. IEEE Computer Society, 571–579.

[5] Matteo Biagiola, Filippo Ricca, and Paolo Tonella. 2017. Search Based Path and Input Data Generation for Web Application Testing. In *International Symposium on Search Based Software Engineering*. Springer, 18–32.

[6] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2019. Diversity-based Web Test Generation. In *Proceedings of 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, 12 pages.

[7] Robert V. Binder. 1996. Testing object-oriented software: a survey. *Software Testing, Verification and Reliability* 6, 3-4 (1996), 125–252.

[8] Cloc 2018. Counts blank lines, comment lines, and physical lines of source code in many programming languages. https://github.com/AlDanial/cloc. (2018).

[9] CoreNLP 2019. Stanford CoreNLP – Natural language software. https://stanfordnlp.github.io/CoreNLP/. (2019).

[10] CRUD 2019. Create, read, update and delete. https://en.wikipedia.org/wiki/Create,_read,_update_and_delete. (2019).

[11] Mark Fewster and Dorothy Graham. 1999. *Software Test Automation: Effective Use of Test Execution Tools*. Addison-Wesley Longman Publishing Co., Inc.

[12] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical Test Dependency Detection. In *Proceedings of the 2018 IEEE Conference on Software Testing, Validation and Verification*. 1–11. DOI: http://dx.doi.org/10.1109/ICST.2018.00011

[13] Z. Gao, C. Fang, and A. M. Memon. 2015. Pushing the limits on automation in GUI regression testing. In *Proceedings of IEEE 26th International Symposium on Software Reliability Engineering (ISSRE '15)*. 565–575.

[14] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 211–222. DOI: http://dx.doi.org/10.1145/2771783.2771784

[15] Google Java Style Guide 2019. Google Java Style Guide. Naming convention for JUnit tests. https://google.github.io/styleguide/javaguide.html#s5.2.3-method-names. (2019).

[16] Google Testing Blog 2014. Testing on the Toilet: Writing Descriptive Test Names. https://testing.googleblog.com/2014/10/testing-on-toilet-writing-descriptive.html. (2014).

[17] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable Testing: Detecting State-polluting Tests to Prevent Test Dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 223–233. DOI: http://dx.doi.org/10.1145/2771783.2771793

[18] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. 2016. WATERFALL: An Incremental Approach for Repairing Record-replay Tests of Web Applications. In *Proceedings of 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, 751–762.

[19] Kim Herzig and Nachiappan Nagappan. 2015. Empirically Detecting False Test Alarms Using Association Rules. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 39–48. http://dl.acm.org/citation.cfm?id=2819009.2819018

[20] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in Continuous Integration: Assurance, Security, and Flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. ACM, New York, NY, USA, 197–207. DOI: http://dx.doi.org/10.1145/3106237.3106270

[21] Sebastian Kappler. 2016. Finding and Breaking Test Dependencies to Speed Up Test Execution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 1136–1138. DOI: http://dx.doi.org/10.1145/2950290.2983974

[22] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the Cost of Regression Testing in Practice: A Study of Java Projects Using Continuous Integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. ACM, New York, NY, USA, 821–830. DOI: http://dx.doi.org/10.1145/3106237.3106288

[23] Wing Lam, Sai Zhang, and Michael D. Ernst. 2015. *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 Science and Engineering, Seattle, WA, USA.

[24] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2013. Capture-Replay vs. Programmable Web Testing: An Empirical Assessment during Test Case Evolution. In *Proceedings of 20th Working Conference on Reverse Engineering (WCRE '13)*. IEEE Computer Society, 272–281.

[25] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2016. Approaches and Tools for Automated End-to-End Web Testing. *Advances in Computers* 101 (2016), 193–237.

[26] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2015. Using Multi-Locators to Increase the Robustness of Web Test Cases. In *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation (ICST '15)*. IEEE, 1–10.

[27] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2016. ROBULA+: An Algorithm for Generating Robust XPath Locators for Web Testing. *Journal of Software: Evolution and Process* (2016), 28:177–204.

[28] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2018. PESTO: Automated migration of DOM-based Web tests towards the visual approach. *Software Testing, Verification And Reliability* 28, 4 (2018).

[29] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale Continuous Testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17)*. IEEE Press, Piscataway, NJ, USA, 233–242. DOI: http://dx.doi.org/10.1109/ICSE-SEIP.2017.16

[30] George A Miller. 1995. WordNet: a lexical database for English. *Commun. ACM* 38, 11 (1995), 39–41.

[31] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179. DOI: http://dx.doi.org/10.1002/spe.2346

[32] Ted Pedersen, Siddharth Patwardhan, and Jason Michelizzi. 2004. WordNet::Similarity: measuring the relatedness of concepts. In *Demonstration papers at HLT-NAACL 2004*. Association for Computational Linguistics, 38–41.

[33] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. 2001. Prioritizing Test Cases For Regression Testing. *IEEE Trans. Softw. Eng.* 27, 10 (Oct. 2001), 929–948. DOI: http://dx.doi.org/10.1109/32.962562

[34] Selenium 2018. SeleniumHQ Web Browser Automation. http://www.seleniumhq.org/. (2018). Accessed: 2017-08-01.

[35] A. Shi, A. Gyori, O. Legunsen, and D. Marinov. 2016. Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications. In *Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST '16)*. 80–90. DOI: http://dx.doi.org/10.1109/ICST.2016.40

[36] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2017. APOGEN: Automatic page object generator for web testing. *Software Quality Journal (SQJ)* 25, 3 (01 Sep 2017), 1007–1039. DOI: http://dx.doi.org/10.1007/s11219-016-9331-9

[37] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. Visual Web Test Repair. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*. ACM.

[38] TEDD 2019. Web Test Dependency Detection using NLP. https://github.com/matteobiagiola/FSE19-submission-material-TEDD. (2019).

[39] Arash Vahabzadeh, Andrea Stocco, and Ali Mesbah. 2018. Fine-grained Test Minimization. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, 210–221. DOI: http://dx.doi.org/10.1145/3180155.3180203

[40] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. 2002. Refactoring Test Code. In *Extreme Programming Perspectives*. Addison-Wesley, 141–152. http://www.cwi.nl/~arie/papers/xp2001.pdf

[41] Zhibiao Wu and Martha Palmer. 1994. Verbs semantics and lexical selection. In *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 133–138.

[42] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muslu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 385–396. DOI: http://dx.doi.org/10.1145/2610384.2610404