WATERFALL: An Incremental Approach for Repairing Record-Replay Tests of Web Applications

Mouna Hammoudi, Gregg Rothermel
University of Nebraska - Lincoln, USA
{mouna,grother}@cse.unl.edu

Andrea Stocco
Università di Genova, Italy
andrea.stocco@dibris.unige.it

ABSTRACT

Software engineers use record/replay tools to capture use case scenarios that can serve as regression tests for web applications. Such tests, however, can be brittle in the face of code changes. Thus, researchers have sought automated approaches for repairing broken record/replay tests. To date, such approaches have operated by directly analyzing differences between the releases of web applications. Often, however, intermediate versions or commits exist between releases, and these represent finer-grained sequences of changes by which new releases evolve. In this paper, we present WATERFALL, an incremental test repair approach that applies test repair techniques iteratively across a sequence of fine-grained versions of a web application. The results of an empirical study on seven web applications show that our approach is substantially more effective than a coarse-grained approach (209% overall), while maintaining an acceptable level of overhead.

CCS Concepts

•**Software and its engineering** → *Software verification and validation;*

Keywords

test case repair, web applications, record/replay tests

1. INTRODUCTION

Record/replay tools enable software engineers to automate the testing of web applications. Record/replay tools capture a set of inputs and actions (mouse clicks, keyboard entries, navigation commands, etc.) that are applied to a web application by a software engineer. During playback, these tools re-deliver the sequence of captured inputs and actions (hereafter referred to as "tests") to the browser engine. The number of record/replay tools used in both research and commercial realms (e.g., CoScripter [19], Jalangi [29], Sahi [26], Selenium [27], Sikuli [37]), and Watir [28]) attest to their importance and popularity.

Unfortunately, tests created by record/replay tools can easily stop functioning as applications evolve [13]. Changes as simple as repositioning page elements or altering the selections in a drop-down list can cause such tests to break. This can greatly limit engineers' abilities to perform regression testing. For this reason, researchers have recently begun devising techniques for automatically repairing record/replay tests [5, 17]. While these techniques can be successful at performing repairs, as we shall show, in real-case scenarios they often are not.

Record/replay tests may break for a variety of reasons, but researchers have singled out "locators" as being particularly problematic [5, 16, 17, 36]. Locators are used by high-level languages such as JavaScript, and by record/replay tools, to identify and manipulate elements on web application GUIs. As web applications evolve, locators are likely to change, and prior instances of locators become obsolete. This causes tests relying on such locators to break. While claims about the problem of locator fragility in the papers just cited are primarily anecdotal, in a recent study [13] we evaluated test breakages[1] that occurred in Selenium IDE tests across 453 versions of eight web applications, and used the results to create a taxonomy of the causes of test breakages.[2] We discovered 1065 individual instances of test breakages, and found that 73.62% of them were related to obsolete locators within tests. For this reason, in this work we focus on issues related to the *repair of locators* in record/replay tests.

Prior work on repairing record/replay tests has focused on situations in which a test t functions on a release R of web application A, but breaks on the subsequent release R' of A. Current techniques for repairing these tests [5, 17] analyze differences between R and R' in an attempt to select an appropriate repair. Code repositories, however, routinely make applications accessible in the form of *finer-grained* intermediate increments (e.g., lower-level versions or commits), so versions of A between R and R' are often available. By applying test repair techniques iteratively across these finer-grained intermediate versions, we may be able to repair tests more effectively than when we apply the same techniques across coarser-grained sets of changes. The additional applications of repair techniques, however, will entail additional costs, and thus, both the effectiveness and the efficiency of fine-grained approaches must be assessed relative to those of coarse-grained approaches.

[1] We define a *test breakage* as the event that occurs when a test that used to function on a web application ceases to be applicable to a new release of that application due to changes that cause the test to halt prematurely. Test breakages differ from "test failures"; these occur when tests continue to function on a new release up to a point at which an oracle signals that a program failure has occurred. In current practice, engineers who execute tests on a new release must distinguish test breakages from test failures; in this work we focus only on tests that have been determined to have broken.

[2] While the results presented in [13] motivate the need for automated test repair techniques, the paper does not present or investigate any particular techniques.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...
http://dx.doi.org/10.1145/2950290.2950294

In this paper we present an approach, WATERFALL, that incrementally repairs tests by considering code changes across finer-grained increments of evolving web applications. While any test repair technique could be plugged in to WATERFALL, in this paper we make use of the locator repair component of the non-incremental capture/replay test repair technique, WATER, presented in [5]. We empirically study our approach on seven open-source web applications for which we had previously created record/replay tests using Selenium IDE, on which 717 test breakages related to test locators had previously been discovered [13]. In our study, applying a repair technique at finer granularities increased its effectiveness (across all versions of all seven web applications) by 209% over coarser granularities. Our results also show that, while WATERFALL required more execution time than the coarse-grained repair approach overall, the overhead it imposed was not excessive. In fact, that overhead was more than compensated for by the reduction achieved in the number of times that humans must intervene to manually repair test cases that could not be automatically repaired.

The contributions of this work are as follows:

- a novel algorithm, WATERFALL, for applying test repairs incrementally across fine-grained releases of web applications;
- an implementation of our algorithm;
- an empirical study that shows that WATERFALL can be effective when applied to several non-trivial open-source web applications, without imposing excessive overhead.

2. BACKGROUND

2.1 Record/Replay Tools

Record/replay tools allow test engineers to capture sequences of inputs and actions applied to a web application's GUI. The recording process creates a test script that can then be replayed in an unattended mode. There are many record/replay tools for web applications available; in this work we utilize Selenium [27], one of the flagship open-source test automation tools for web applications.

Figure 1 (top) depicts a typical sample web application, allowing users to login. Figure 1 (bottom) shows the associated HTML code, consisting of a form, two input fields for entering a username and password, and a submit input field for submitting credentials. Table 1 shows a test created by Selenium IDE via the application of a sequence of inputs and actions to the login web page. Each input or action causes a *Selenese* command to be inserted into the test. Each *Selenese* command is denoted by a tuple: <*action, locator, value*>. The action component indicates either an event that is performed on the user interface during recording, or an action specific to Selenium's control of the replay. The locator component specifies the web element the user is interacting with during a step of the recording process. The value component refers to any input entered by the user within the element specified by the locator.

2.2 The WATeR Test Repair Technique

While any technique could be "plugged in" to our test repair approach, for this work we chose the WATER (Web Application Test Repair) technique, created by Choudhary et al. [5]. We chose WATER because it has a procedure geared specifically towards locator repair. While there exists one other approach for improving locator robustness [16], that approach specifically targets only XPath locators, and applying it to the web applications used in our study would require us to convert all other types of locators in our recorded tests to XPath locators, rendering our results less generalizable.

At a high level, WATER is a differential testing technique used to compare the executions of a test t over two different releases R and R' of a web application under test, where t runs properly on R

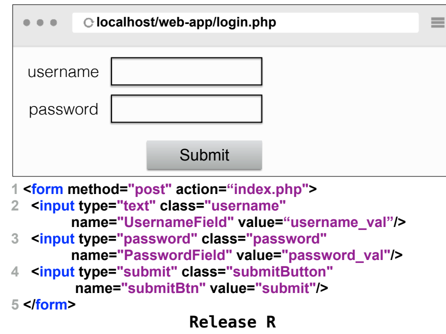

Figure 1: Sample web application showing a login form (top) and the corresponding HTML code (bottom)

Table 1: A Selenium Test for the Web Application of Figure 1

Stmt	Action	Locator	Value
1	open	localhost/web-app/login.php	
2	type	name=UsernameField	"jsmith"
3	type	name=PasswordField	"secret123!"
4	click	//form/input[3]	

but breaks on R'. WATER executes t on R and R' and gathers data about these executions. WATER then examines the differences between R and R', and based on these differences and execution data, uses heuristics to find a set of potential repairs PR for t. For each potential repair p in PR, WATER executes t using p. If t runs properly on R', p is added to a list of *suggested repairs*. This step is repeated until each potential repair in PR has been considered. The list of suggested repairs is then output to developers for their consideration. The list of suggestions is ordered according to a set of heuristics, such that earlier suggestions have higher probabilities of yielding a correct repair. Choudhary et al. [5] also explain that if a broken test statement works correctly after a potential repair is applied, but the test encounters a breakage again at a later statement, WATER is rerun to attempt to repair the new breakage. Space limitations prevent us from providing a more detailed description of the algorithm, but details are available in Choudhary et al.'s paper [5].

WATER can suggest locator repairs that are not, in fact, appropriate. Thus, Choudhary et al. state that their technique "suggests" test repairs to engineers, rather than simply applying the repairs and assuming that one is correct if the test runs to completion. It also seems clear that Choudhary et al. present suggested repairs for t to engineers only after t has run to completion, not at an earlier point p in t at which a repair has been suggested. Finally, WATER is not guaranteed to find a test repair. In a case study presented in their paper, Choudhary et al. show instances in which WATER fails, instances in which it suggests inappropriate repairs, and instances in which it suggests appropriate repairs.

3. A TEST REPAIR SCENARIO

Let R and R' be releases of web application A, and let t be a test that runs properly on R but breaks on R'. A *coarse-grained* approach to repairing t attempts to do so by applying a repair technique to (R, R', t), and if successful, produces a repaired test t_{fix}. Figure 2 depicts this approach.

In the foregoing scenario, let $V_1, V_2, \ldots V_k$ be intermediate versions or intermediate commits created as R evolved into R'. A fine-grained approach to repairing t attempts to execute t on the sequence of intermediate versions or commits between R and R'. If a breakage occurs in some intermediate version V_k, the repair technique is applied to (V_{k-1}, V_k, t). If successful, the approach produces a test that no longer breaks on V_k. The approach then

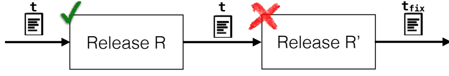

Figure 2: Repairing t using a coarse-grained approach

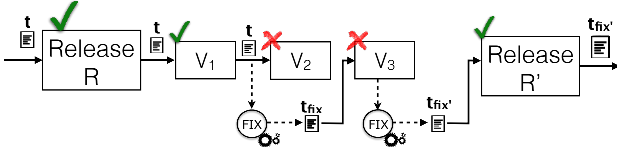

Figure 3: Repairing t using a fine-grained approach

iterates through subsequent intermediate versions or commits, repairing t where necessary, until it has produced a repaired version of t that runs correctly and to completion on R'.

Figure 3 shows a case in which the fine-grained approach is applied where three intermediate releases are available between R and R'. In the figure, t passes on the initial release R. Then, t is transferred forward to the next version, V_1, where it passes and is transferred forward to V_2. When t is executed on V_2 a breakage occurs and t needs to be corrected. The repair technique is applied to (V_1, V_2, t) and outputs t_{fix}, which is transferred forward to the next version, V_3. Here, t_{fix} fails and must be repaired again. The repair technique is applied to (V_2, V_3, t_{fix}) and outputs $t_{fix'}$, which is transferred forward to the last version, R', where it passes.

We believe that the coarse-grained approach will be less effective at repairing tests than the fine-grained approach. The number of code changes between R and R' can be far larger than the number of changes between any pair of successive releases or commits. The changes between R and R' may also be intertwined in manners that render repair heuristics less effective. Applying a repair technique iteratively to intermediate versions or commits should leave the technique with fewer, less intertwined changes to consider per application, increasing its chances of success.

On the other hand, a fine-grained approach may need to apply repairs many more times than a coarse-grained approach, increasing its overall cost. This can occur, for example, if a specific locator breaks, for different reasons, across several intermediate versions. Consider Figure 3, and suppose that the test breakage in V_2 is caused by the fact that an attribute locator has been changed from "name = Name" to "name = NewerName", and suppose that the test breakage in V_3 is caused by the fact that the same locator has been changed again to "name = NewestName". In this case the coarse-grained approach would require only one application, to change the locator used in R ("name = Name") to one that works in R' ("name = NewestName"), whereas the fine-grained approach would be applied twice, i.e., iteratively on each problematic intermediate version. In this case, the extra cost associated with extra applications of the test repair technique across all versions may exceed the cost of applying the technique once on R and R'.

An additional cost factor concerns the manual effort within the test repair process itself. WATER does not always succeed, and when it fails, engineers need to intervene and find repairs manually. Given that the fine-grained approach may require more applications of a test repair technique than the coarse-grained approach, the fine-grained approach may also require more manual repairs.

4. THE WATERFALL APPROACH

4.1 A Basic Algorithm

A fine-grained test repair algorithm operates on releases R and R' of web application A, for which a sequence of intermediate versions between R and R' are available. Given test t, the algorithm proceeds until either (1) it reaches release R' and t passes and runs properly on it, or (2) t breaks on some intermediate version V_k prior to R'. In this case, the algorithm attempts to repair t_{k-1} (the test that reaches V_k) such that it operates correctly on V_k. This process iterates until (1) occurs or until the algorithm unable to repair a test at some point in the sequence.

Algorithm 1 presents WATERFALL, our fine-grained test repair algorithm. WATERFALL accepts releases R and R', and a test t that ran properly on R. WATERFALL initially obtains the list of versions (VersionList) between R and R' (Line 2), including R and R' at the beginning and end of the list, respectively. It then iterates through VersionList (Lines 4-17). For each version V_{next} considered, WATERFALL attempts to run (possibly previously repaired) test t on that version. If t does not break, WATERFALL begins a new iteration on the next version, retaining t. If t does break, WATERFALL attempts to repair it (procedure `RepairTest`, Line 9). If `RepairTest` fails to repair t, it returns $null$ and WATERFALL terminates. If the repair succeeds, WATERFALL begins a new iteration on the next version, promoting the repaired test, t_{fixed}, to t. If WATERFALL reaches the end of the sequence of versions, R', and is able to run t on that version (or repair t so that it runs on that version), it returns t.

Algorithm 1 Fine-grained Test Repair Algorithm

```
1: function WATERFALL(R, R', t) : t_fixed
2:     VersionList ← GetVersions(R, R')
3:     V ← R
4:     for i ← 0 to VersionList.length() do
5:         V_next ← VersionList.get(i + 1)
6:         if RunTest(V_next, t).passes() == true then
7:             V ← V_next
8:         else
9:             t_fixed ← RepairTest(V, V_next, t)
10:            if t_fixed == null then
11:                return null
12:            else
13:                t ← t_fixed
14:                V ← V_next
15:            end if
16:        end if
17:    end for
18:    return t
19: end function
```

Note that `RepairTest` could be instantiated using any test case repair algorithm; in this work we instantiate it using our implementation of WATER. Further, if WATERFALL is unable to repair a test, i.e., returns $null$ (Line 11), there are additional steps that could be taken. A test engineer could intervene and repair the test manually, and then reapply WATERFALL from that point. Alternatively, `RepairTest` could be invoked again to retrieve a different suggested repair. Such an approach makes sense when utilizing WATER, which actually suggests a list of repairs.

4.2 Example

We now illustrate WATERFALL using an example. Let the web application in Figure 1 be R, and suppose that it undergoes a three step evolution process. Figure 4 shows the evolution of the HTML code for three versions, beginning with an intermediate version V_1 (top), then an intermediate version V_2 (middle), and finally a release R' (bottom). Specific changes are enclosed in ellipses.

In V_1, the values of the class and value attributes have been modified for the username and password text fields. Further,

```
1 <form method="post" action="index.php">
2  <input type="text" class="UserClass" name="UsernameField" value="UserValue"/>
3  <input type="password" class="PasswordClass" name="PasswordField" value="PasswordValue"/>
4  <input type="submit" class="SubmitClass" name="submitBtn" value="submit"/>
5 </form>
```

Version V1

```
1 <form method="post" action="index.php">
2  <input type="text" class="UserClass" name="user" value="UserValue"/>
3  <input type="password" class="PasswordClass" name="PasswordField" value="PasswordValue"/>
4  <input type="submit" class="SubmitClass" name="sub" value="submit_val"/>
</form>
```

Version V2

```
1 <form method="post" action="index.php">
2 <div class="login">
3 <input type="text" class="UserClass" name="user" value="UserValue"/>
4 <input type="password" class="PasswordClass" name="PasswordField" value="PasswordValue"/>
5 <input type="submit" class="SubmitClass" name="sub" value="submit_val"/>
6 </div>
7 <div class="developer_info">
 ...
11 </div>
12 </form>
```

Release R′

Figure 4: An Example of Web Application Evolution

the class attribute of the submit button has been modified. For example, in the username text field (Line 2), the class attribute has been changed from class="username" to class="UserClass", and the value attribute has been changed from value="username_val" to value="UserValue". Despite these changes, the test in Table 1 still passes on this version, because in V_1 the locators for the username field, password field, and submit button remain stable.

In V_2, the name attribute of the username text field has changed from name="UsernameField" to name="user". Hence, Statement 2 of the test in Table 1 breaks when executed on V_2. In this case, WATERFALL considers V_1 and V_2 and compares Line 2 of the HTML code in V_1 with Line 2 of the HTML code in V_2. Since all of the attribute values are identical except for the name attribute value, WATERFALL is able to identify the equivalent username input field in V_2 and repair the locator breakage in Line 2.

In R', the developer has modified the page layout by introducing a div (Line 2) that differentiates the "login" portion of the web page from the "developer info" portion. This causes Statement 4 of the test in Table 1 to break, because the XPath locator no longer reflects the hierarchical path identifying the submit button within the DOM tree of the web page under test. In this case, WATERFALL compares Line 4 of the HTML code for V_2 with Line 5 of the HTML code for R'. All the attribute values of the submit button remain unchanged in this case. Thus, WATER is able to recognize the equivalent submit button in R'.

Table 2 displays t as repaired by WATERFALL. The first repair updates the name locator in Line 2 from name=UsernameField to name=user. The second repair updates the XPath locator in Line 4 of the test from //form/input[3] to //form/div/input[3]. WATERFALL is capable of performing these repairs using a fine-grained approach; however, neither of these breakages can be repaired by WATER using a coarse-grained approach. WATER compares the class, name and value attributes of the username and submit button input fields. To repair the breakage in Statement 2 of the test in Table 1 (the username locator breakage), WATER compares Line 2 in R (Figure 1) to Line 3 in R'(Figure 4-bottom). Because the values for the class, name and value attributes have changed, WATER is not able to recognize the username text field in R'. Thus, WATER is not able to suggest any repairs for the username locator breakage.

Table 2: The Repaired Selenium Test

Stmt	Action	Locator	Value
1	open	localhost/web-app/login.php	
2	type	name=user	"jsmith"
3	type	name=PasswordField	"secret123!"
4	click	//form/div/input[3]	

Similarly, to repair the breakage encountered in Statement 4 of the test in Table 1 (the submit button locator), WATER compares Line 4 in R to Line 5 in R'. Because all the values for the class, name and value attributes have changed, WATER cannot recognize the equivalent submit button in R'. Thus, WATER is not able to suggest any repairs for the submit button locator breakage. This example is in fact representative of many of the breakages that we encountered in our empirical study.

The superior performance of WATERFALL on this example is due to the smaller number of changes made to the HTML code across successive versions of the web app. By applying the repairs to smaller sets of changes, WATERFALL is able to identify equivalent HTML input fields across releases. Therefore, WATERFALL is able to suggest correct repairs more often than a coarse-grained approach in which WATER is applied just to R and R'.

4.3 Improvements to the Basic Algorithm

Given (R, R', t), WATER outputs a *set* of *suggested* repairs. Each suggestion allows t to run to completion on R', but this does not mean that t is operating as designed: e.g., it may now exercise a different use case scenario than the one it was intended to exercise. Thus, test engineers may wish to inspect the list of suggestions and select a repair that allows t to run properly.

In our basic algorithm, in contrast, `RepairTest` is applied iteratively across a sequence of versions. This raises two issues which, if addressed, might yield improvements in WATERFALL's effectiveness. To understand these issues, again consider the example presented in Figure 3. Suppose that when `RepairTest` is run on (V_1, V_2, t), it suggests three repairs, S_1, S_2, and S_3. The fact that S_1, S_2, and S_3 are suggested means that each of them yields a test that passes on V_2. Our basic algorithm selects the first suggested repair, S_1, and moves forward to the next version, V_3, transferring t_{fix} to it. In our example, t_{fix} breaks on V_3. Suppose `RepairTest` is now unable to find a repair for t_{fix} on V_3. In this case, our basic algorithm terminates and returns $null$.

The first issue to consider for this scenario is the manner in which the algorithm treats suggestions. If the algorithm were to select one of the other suggestions for repairs at V_2 (S_2 or S_3), then on reaching V_3 it *might* have found a repair there. A potential improvement to the algorithm would allow it to *backtrack* to prior lists of suggestions when it cannot repair a test on a subsequent breakage. In the scenario just presented, a backtracking version of the algorithm, having failed to find a repair at V_3, would return to V_2 and select a new suggested repair, and proceed forward with that.

While theoretically feasible, backtracking versions of WATERFALL could face scalability problems, depending on (1) the sizes of suggestion lists and (2) the number of times that backtracking is called for when processing a sequence of versions. Heuristics might be needed to avoid an exponential explosion in the number of sequences of repairs to consider. Nevertheless, with appropriate use of such heuristics, a backtracking version of WATERFALL might be more effective than our basic version.

The second issue to consider for this scenario is the point at which humans are asked to inspect lists of suggested repairs. WATERFALL assumes that humans will be shown suggested repairs only when R' is reached, when a list of suggested repair sequences is output. Engineers could, however, become involved each time a

breakage is located in a version V_k, and select, from the list of repairs suggested at that version, the repair they consider appropriate. This may also improve the algorithm's effectiveness, at the cost of additional engineer time.

5. EMPIRICAL STUDY

We consider the following research questions:

RQ1: *How do coarse- and fine-grained test repair approaches compare in terms of effectiveness?*

RQ2: *How do coarse- and fine-grained test repair approaches compare in terms of efficiency?*

In our study, the coarse-grained approach is represented by WATER, and the fine-grained approach is represented by WATERFALL. In this study we utilize our basic algorithm, on the view that it makes sense to first determine whether that algorithm has promise prior to implementing more complex algorithms. We do provide insights into the use of backtracking, however, in Section 6.3.

5.1 Objects of Analysis

5.1.1 Web Applications

As objects of analysis we chose several web applications utilized in our earlier study of test breakages in record/replay tests [13]. The applications selected there were required to: (i) have at least 20 installable and executable versions/commits, (ii) have at least 30,000 lines of code, (iii) have been downloaded at least 5,000 times, and (iv) have experienced at least 300 commits. Requirement i is crucial to application usability; the other requirements lessen threats to external validity by ensuring that the selected applications are non-trivial and widely used.

Table 3 provides data on the web applications we used, including their names, the number of releases, versions, and commits we used, the number of lines of code they contained (counted using `cloc`[3] and averaged across the versions), the number of times they had been downloaded, and the number of tests used for them. *PHPAddressBook* helps users manage and organize contacts. *PHPAgenda* lets users manage calendars, schedule appointments, holidays, todo lists, and share them with other users. *PHPFusion* is a content management system that helps users create, manage and administer a web site without knowledge about web programming. *Joomla* is a content management system that helps users with little experience in web programming publish content. *MyCollaboration* is a collaboration platform that helps users manage customer information and projects. *Dolibarr* is used for enterprise resource planning and customer relationship management. *YourContacts* is used by companies to manage their contacts. While all applications use JavaScript, HTML, MySQL, and CSS, *MyCollaboration* is written in Java, and the other applications are written in PHP.

5.1.2 Releases, Versions, and Commits

For each web application considered, we required a way to investigate test repair at both the coarse- and fine-grained levels. The ways in which we were able to do this varied, however, across the applications, because some provide (**case 1**) commits between versions, while others provide (**case 2**) only various levels of versions. In both of these cases there are scenarios in which coarse- and fine-grained test repair are viable. In case 1, coarse-grained repair could be applied across pairs of successive versions, whereas fine-grained repair could be applied across pairs of intervening commits, provided these commits result in executable instances of the web applications. In case 2, coarse-grained repair could be applied across pairs of successive higher-level releases, while fine-grained repair could be applied across pairs of intervening versions. Such an approach could make sense in practice if regression testing is restricted to higher-level releases, as might happen if the time available for testing lower-level releases were constrained. To provide a uniform way to describe the two cases, we refer to the coarser-granularity versions as *releases* and we refer to the finer-granularity versions (whether commits or intermediate versions) as *versions* or (on occasions where clarity requires) *intermediate versions*.

YourContacts and *Dolibarr* were available at the levels of releases and commits. For *YourContacts* we were able to utilize 12 releases, with 2–23 commits between each pair. For *Dolibarr* we were able to utilize eight releases. In this case, however, there were an enormous number of commits between each pair of releases, many of which involved no code changes, so we selected every 75th commit between each pair. For the other five web applications (*PHPAddressBook*, *PHPAgenda*, *PHPFusion*, *Joomla*, and *MyCollaboration*), commits were not available, but on each, there were three levels of versions (major, minor, patch); this allowed us to choose several versions as releases with several finer-grained versions as intermediate versions between them. This choice is motivated by empirical evidence that a larger percentage of breakages occur at intermediate release levels [25] than at higher levels. For each of these web applications we chose, as releases, all major (e.g, 1.0 or 2.0) or minor (e.g, 1.1.0, 1.2.0) releases. As intermediate versions we selected the patch versions between successive major or minor releases, yielding sequences such as (1.0, 1.0.1, 1.0.2, 1.1) or (1.1.0, 1.1.1, 1.1.2, 1.2.0), respectively. We disregarded cases in which only zero or one patch versions were available.

In either of the foregoing cases, we obtain, for each web application, several of what we refer to as *sequences of versions*, which begin with a release, end with a subsequent release, and contain two or more intermediate versions between these releases. For example, we obtain sequences of versions of the form (R, V_1, ..., V_{n-1}, V_n, R'), where R and R' are releases, and V_1, ..., V_n are intermediate versions (or commits). Table 3 lists the number of releases, and the number of intermediate versions (or commits), that we ultimately retained, for each web application considered.[4]

Table 3: Objects of Analysis

ID	Web App Name	Releases	Versions / Commits	LOC	Downloads	Tests
A1	PHPADDRESSBOOK	10	74	35,675	126,146	44
A2	PHPAGENDA	6	34	43,831	64,605	42
A3	PHPFUSION	6	50	256,899	1,605,195	47
A4	JOOMLA	8	92	312,978	>50,000,000	56
A5	MYCOLLABORATION	6	30	116,345	7,638	39
A6	DOLIBARR	8	21	42,010	864,698	38
A7	YOURCONTACTS	12	88	64,765	676,543	57

5.1.3 Record/Replay Test Suites

The record/replay tests used in this study were initially created in the context of our earlier work [13]. We created these tests because no appropriate tests were available for the web applications utilized.[5] To create these tests we followed a systematic, iterative procedure for each web application, creating tests that achieved coverage of use cases and exceptional behaviors. Such use-case-based approaches to testing systems at the interface level are common in practice, and well-suited to the use of record/replay tools.

[3]cloc.sourceforge.net

[4]For a list of the actual releases, versions, and commits that we considered and pointers to the source code repositories for each object of study, see https://sites.google.com/site/repairrecordreplaytests.

[5]*Joomla* does include 42 WebDriver tests, but no IDE tests, and only for its first version.

Given a test suite T created for a version V_k the first author followed a three-step process – a process that WATERFALL needs to replicate, with some differences, in this current study, as described in Section 5.3. In Step 1, she executed T on the next version of the web application, V_{k+1}, and noted each case in which a test broke. In Step 2, she manually repaired each of the tests that were repairable – an iterative process because repairing one test breakage might allow that test to proceed further and break again later. In Step 3, she added new tests covering new functionality to the test suite. This resulted in a new test suite T' that now functioned on V_{k+1}. She then repeated these three steps for each subsequent version of the web application until each had been tested and the causes of all test breakages had been noted. The numbers of tests listed in Table 3 are the numbers available on the final version of each web application considered.

5.2 Variables and Measures

5.2.1 Independent Variables

Our independent variable is the test repair approach used: coarse-grained (i.e., WATER) or fine-grained (i.e., WATERFALL).

5.2.2 Dependent Variables

Our goal is to assess the effectiveness and efficiency of the test repair approaches.

Effectiveness. To measure the effectiveness of the coarse-grained repair approach, we first count C_{tot}, the total number of repairs required when applying the approach to (R, R', t); this number equals the total number of locator breakages of t on R'. We then count C_{rep}, the total number of locator breakages of t that WATER is able to correctly repair.

To compare the overall effectiveness of the coarse- and fine-grained repair approaches, it is not appropriate to count breakages encountered by the fine-grained approach at intermediate versions, because such breakages may "cancel each other out". Section 3 provided an example of this in which an attribute locator is first changed, in some version, from "name=Name" to "name=NewerName", and then changed, in a subsequent version, to "name = NewestName". In this case, the fine-grained repair approach encounters each of these changes separately, and must repair t for each of them independently. As far as the coarse-grained repair approach is concerned, however, the only visible breakage involves the fact that "Name", used in R, is changed to "NewestName" in R', and only one repair is needed to address the resultant breakage. It is this repair, and others that the coarse-grained repair approach encounters, that matters for the sake of overall comparisons of the techniques.

Thus, to compare the overall effectiveness of the coarse- and fine-grained test repair approaches, we consider how well the fine-grained approach does at repairing *just those breakages that the coarse-grained approach needs to repair at* R'. More precisely, we let F_{tot}, the total number of breakages that need to ultimately be repaired by WATERFALL in R', be equal to C_{tot}. Then, we count F_{rep}, the total number of breakages out of F_{tot} that WATERFALL was ultimately able to correctly repair – i.e., breakages no longer present in R' following the application of the algorithm.

The methods just described require a mechanism for determining whether WATER is able to correctly repair a breakage. To perform this determination, each time WATER produced a test t' that executes without breaking on R', we manually inspected t' to ascertain whether it achieved the original goal for t, and pronounced it "correct" only if it did. When applying WATERFALL, each time it produced a test t' that executes on all versions in a sequence without breaking, we performed an analogous inspection.

Efficiency. We utilize two measures of efficiency. First, we measure the efficiency of the approaches in terms of *execution time*. Since our algorithms and instrumentation code are implemented in Java, we used a Java subroutine, java.lang.System.currentTimeMillis(), which returns the current time in milliseconds. We invoked this subroutine before and after calls to the coarse- and fine-grained repair procedures (this allowed us to avoid measuring other computational load such as CPU contention and memory saturation). In this way, we were able to derive the current time before and after the calls, and compute run times by subtracting the second from the first. Note that there is a complicating factor in this process involving unrepaired intermediate breakages encountered when applying the fine-grained repair approach, that we must account for when measuring its execution time. We discuss this in Section 5.3.

A second efficiency metric, as noted in Section 4.3, relates to the additional costs for checking repairs, and finding and applying repairs in cases where automated repair techniques fail. Without conducting an empirical study of humans we cannot assess this cost in terms of time. However, studies involving humans are expensive. Before incurring that expense it makes sense to assess whether our approach has prospects for success using a dependent variable that serves as a proxy for the time spent by humans. As such a proxy, we measure the *number of manual repairs required* by each approach. To do this for the coarse-grained repair approach, we simply count the number of unrepaired breakages that remain in R': this is given by $C_{tot} - C_{rep}$. To do this for the fine-grained repair approach there are two sets of breakages to count: (1) breakages that remain in R', and (2) breakages that occur in intermediate versions that cannot be repaired by WATERFALL and must be repaired by engineers before the process can continue. The former is given by $F_{tot} - F_{rep}$, and the latter is given by $F_{int_{done}} - F_{int_{needed}}$, where $F_{int_{done}}$ is the sum of repairs made by WATERFALL at each intermediate version between R and R' and $F_{int_{needed}}$ is the sum of repairs needed at each intermediate version between R and R'.

5.3 Study Operation

For each web application A, we considered each pair of successive releases (Ri, Rj) of A and performed the following activities.

First, for each test t applicable to Rj, we applied WATER to (Ri, Rj, t), which resulted in a list S of suggested repairs for t. Note that the list of locator breakages in Rj is already known (having been determined in our earlier study [13]), and the size of that list is C_{tot}. Each time WATER selected a locator repair from S, the first author inspected it and determined, for each of the known locator breakages in Rj, whether the repair applied by WATER was correct. The number of correct locator repairs selected from S and applied by WATER is equal to C_{rep}.

Second, for each test t applicable to Rj, we applied WATERFALL to the sequence of intermediate versions beginning with Ri and ending at Rj. This process was more complex for two reasons: (i) when the fine-grained approach is applied to a sequence of versions, WATERFALL can encounter, at some intermediate version, a locator breakage that it is unable to repair, and (ii) our tests do contain some breakages due to causes not involving locators that our current implementation of WATERFALL cannot repair. In practice, in either of these cases, an engineer would need to intervene, address the cause of the breakage, and repair the test manually. In our study, we simulate that practice. (We say "simulate" because we already know where all the breakages are and what repairs are needed to correct them, given the data from our prior study [13], and we make use of that knowledge to apply known repairs.)

Our simulation is similar to the three step process, by which we evolved test suites for our web applications. To describe the process

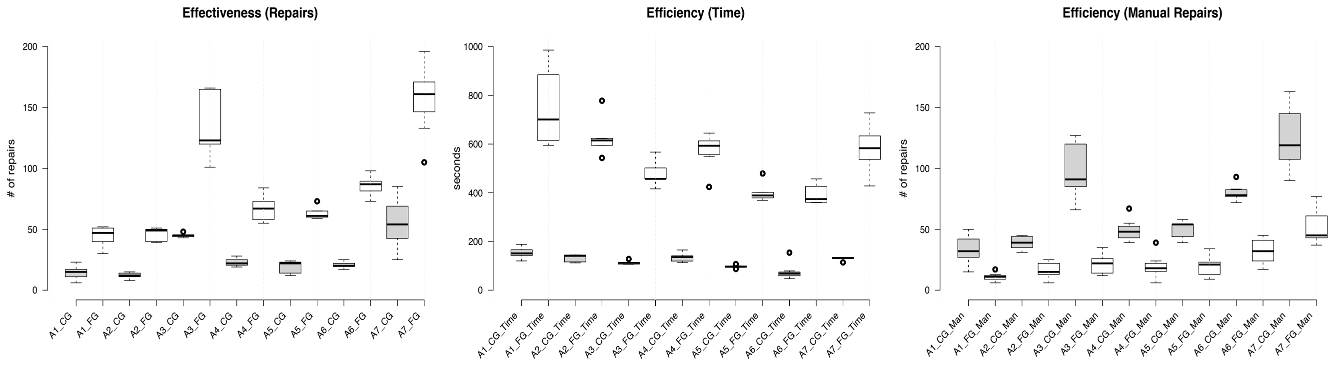

Figure 5: Effectiveness, efficiency, and manual repair results, all applications, all sequences of versions.

we refer to the scenario in Figure 3. We initialize F_{tot} and F_{rep} to zero. Suppose the breakage encountered on V_2 is a locator breakage WATERFALL can repair, and the breakage encountered in V_3 is a breakage (locator or other) WATERFALL cannot repair. We first apply WATERFALL to (V_1, V_2, t); WATERFALL succeeds, creating repaired test t_{fix}, and we increment both F_{tot} and F_{rep} by one. We next apply WATERFALL to (V_2, V_3, t_{fix}), and WATERFALL is unable to repair the breakage. Whether the breakage involves a locator or not, we manually apply the repair utilized in our first study, creating repaired test $t_{fix'}$, and increment F_{tot} (but not F_{rep}) by one. Next, we apply WATERFALL to (V_3, R', t'_{fix}), and t'_{fix} passes in R' and performs its intended behavior. Thus, t has been successfully repaired by WATERFALL for the sequence of versions R-R'. In this example, V_3 is the last intermediate version in the sequence R-R', so we do not need any further repairs. If there were additional intermediate versions between V_3 and R', we would apply WATERFALL iteratively to each subsequent version that triggers a test breakage, adjusting F_{tot} and F_{rep} appropriately, until no additional test breakages in subsequent versions arise.

If multiple breakages arise when running t on an intermediate version, the foregoing process becomes more complicated but continues to function. We apply WATERFALL to each locator breakage in t and note the locator breakages in t that it can and cannot repair, incrementing F_{tot} and F_{rep} accordingly. We then manually correct any remaining unrepaired breakages.

In Section 5.2, when discussing our approach for measuring the run time of the fine-grained approach, we mentioned that there was a complicating factor in that process. That complicating factor arises in cases such as those just described, where a breakage is encountered that WATERFALL cannot repair. In these cases, we keep track of the execution time spent by WATERFALL on the sub-sequence of versions up to the problematic breakage, and then also measure execution times on following sub-sequences. The run time of the fine-grained approach is given by the sum of the elapsed times across all sub-sequences.

5.4 Threats to Validity

External validity threats concern the generalization of our findings. We considered only seven web applications and our results may not generalize to others. The selection criteria that we had adopted in choosing these, however, did ensure that they were of non-trivial size and had multiple versions, and had experienced many downloads and commits. As a second threat, our results are gathered relative to tests repaired using WATER and WATERFALL, and generalization to other repair approaches is not possible.

Internal validity threats concern uncontrolled factors that may have affected our results. The first author created the tests. This task, however, requires reasoning that cannot be automated, so it is difficult to envision less threat-prone approaches. Moreover, to reduce the subjectivity involved, she followed a systematic and structured procedure. Finally, all of these tasks were performed during our prior study [13], and prior to our having envisioned the approach and the study presented in this paper. This reduces potential sources of bias that could have occurred if the tasks had been performed specifically with this study in mind.

Construct validity threats concern our metrics and measures. We measure the numbers of breakages found and repaired, but differences in these numbers across the two approaches do not necessarily correlate with differences in the amount of effort needed to manually repair breakages that remain. Similarly, we measure the cost of the approaches in terms of the machine time required to run techniques and the number of manual repairs required, but this may not directly correlate with the time that may be spent, by engineers, repairing breakages that are not automatically repaired. We return to this latter point, however, in Section 6, where we are able to offer some additional insights.

5.5 Results and Analysis

Figure 5 presents the data from our study graphically, in the form of boxplots. From left to right, the three graphs present results for effectiveness, efficiency in terms of technique execution time (in seconds), and efficiency in terms of numbers of manual repairs required, respectively. In each graph, results are shown for each of the seven web applications as a pair of boxplots. Each pair of boxplots represents the datasets gathered for the coarse- and fine-grained approaches, respectively, across all of the sequences of versions considered for the given web application. For space considerations, web applications are identified by their IDs as presented in Table 3. Thus, for example, the first two boxes in the leftmost figure represent the distributions of the effectiveness values for web application A1 (*PHPAddressBook*), for the coarse-grained (first box) and fine-grained (second box) test repair techniques, across the nine sequences of versions available for that application.

The boxplots show that for every web application, (1) the fine-grained repair approach is more effective than the coarse-grained approach, (2) the fine-grained approach requires more time than the coarse-grained approach, and (3) the fine-grained approach requires fewer manual repairs than the coarse-grained approach.

5.5.1 RQ1: Effectiveness

Table 4 presents effectiveness results for each web application, for each sequence of versions or commits utilized. In the column headers, "CG" is an acronym for "coarse-grained" and "FG" is an acronym for "fine-grained". Column 1 lists the web applica-

Table 4: Effectiveness and Efficiency Results Across all Sequences of Versions, for all Web Applications

Web Applications			Effectiveness						Efficiency					
Name	**Versions**	**# releases**	**CG Repairs Done** (C_{rep})	**FG Repairs Done** (F_{rep})	**Total Repairs Needed** ($C_{tot} = F_{tot}$)	**Diff and Increase (%)**	**FG Int. Repairs Done** ($F_{int_{done}}$)	**FG Int. Repairs Needed** ($F_{int_{needed}}$)	**CG Time (s)**	**FG Time (s)**	**Diff and Overhead (%)**	**CG Man** C_M	**FG Man** F_M	**Diff and Decrease (%)**
PHPAddressBook	R1-R2	18	10	47	59	37 (370%)	65	70	144	595	450 (312%)	49	17	32 (-65.31%)
	R2-R3	9	15	48	53	33 (220%)	59	67	156	595	439 (281%)	38	13	25 (-65.79%)
	R3-R4	5	6	52	56	46 (767%)	63	70	151	615	464 (308%)	50	11	39 (-78.00%)
	R4-R5	14	11	51	53	40 (364%)	61	68	120	624	505 (421%)	42	9	33 (-78.57%)
	R5-R6	3	18	47	50	29 (161%)	60	67	166	781	615 (369%)	32	10	22 (-68.75%)
	R6-R7	5	23	52	55	29 (126%)	62	71	177	885	709 (402%)	32	12	20 (-62.50%)
	R7-R8	3	15	40	42	25 (167%)	59	65	188	701	514 (274%)	27	8	19 (-70.37%)
	R8-R9	6	16	33	35	17 (106%)	48	58	142	986	844 (594%)	19	12	7 (-36.84%)
	R9-R10	11	17	30	32	13 (76%)	50	54	121	888	767 (634%)	15	6	9 (-60.00%)
Total for PHPAddressBook		**74**	**131**	**400**	**435**	**269 (205%)**	**527**	**590**	**1363**	**6670**	**5306 (389%)**	**304**	**98**	**206 (-67.76%)**
PHPAgenda	R1-R2	3	12	39	43	27 (225%)	67	78	144	595	450 (312%)	31	15	16 (-51.61%)
	R2-R3	9	11	51	56	40 (364%)	69	86	144	615	470 (326%)	45	22	23 (-51.11%)
	R3-R4	5	15	50	54	35 (233%)	65	67	141	623	483 (344%)	39	6	33 (-84.62%)
	R4-R5	3	14	49	58	35 (250%)	76	80	112	778	666 (595%)	44	13	31 (-70.45%)
	R5-R6	14	8	40	43	32 (400%)	62	84	116	543	427 (367%)	35	25	10 (-28.57%)
Total for PHPAgenda		**34**	**60**	**229**	**254**	**169 (282%)**	**339**	**395**	**658**	**3154**	**2497 (380%)**	**194**	**81**	**113 (-58.25%)**
PHPFusion	R1-R2	6	44	101	110	57 (130%)	204	230	112	457	345 (307%)	66	35	31 (-46.97%)
	R2-R3	20	43	123	134	80 (186%)	210	213	113	457	344 (305%)	91	14	77 (-84.62%)
	R3-R4	9	45	166	172	121 (269%)	215	221	107	502	395 (368%)	127	12	115 (-90.55%)
	R4-R5	8	45	120	130	75 (167%)	186	198	106	416	311 (295%)	85	22	63 (-74.12%)
	R5-R6	7	48	165	168	46 (244%)	204	227	128	567	439 (344%)	120	26	94 (-78.33%)
Total for PHPFusion		**50**	**225**	**675**	**714**	**450 (200%)**	**1019**	**1089**	**566**	**2401**	**1834 (324%)**	**489**	**109**	**380 (-77.71%)**
Joomla	R1-R2	26	23	67	78	44 (191%)	76	89	125	593	469 (376%)	55	24	31 (-56.36%)
	R2-R3	35	20	84	87	64 (320%)	79	92	138	594	456 (330%)	67	16	51 (-76.12%)
	R3-R4	4	21	73	65	42 (200%)	89	105	144	568	423 (293%)	44	18	26 (-59.09%)
	R4-R5	9	22	56	70	34 (155%)	99	114	114	548	434 (381%)	48	39	19 (-39.58%)
	R5-R6	5	19	55	61	36 (189%)	131	145	136	424	288 (212%)	42	20	22 (-52.38%)
	R6-R7	5	28	60	67	32 (114%)	105	113	113	632	519 (460 %)	39	15	24 (-61.54%)
	R7-R8	8	27	73	77	46 (170%)	143	145	165	645	480 (290%)	50	6	44 (-88.00%)
Total for Joomla		**92**	**160**	**458**	**505**	**298 (186%)**	**722**	**803**	**935**	**4004**	**3069 (328 %)**	**345**	**128**	**217 (-62.90%)**
MyCollaboration	R1-R2	3	12	61	66	49 (408%)	87	103	99	389	290 (293%)	54	21	33 (-61.11%)
	R2-R3	5	23	59	62	36 (157%)	96	102	107	379	273 (256%)	39	9	30 (-76.92%)
	R3-R4	7	14	65	72	51 (364%)	98	114	96	479	382 (397%)	58	23	35 (-60.34%)
	R4-R5	11	24	73	78	49 (204%)	90	119	96	369	272 (283%)	54	34	20 (-37.04%)
	R5-R6	4	22	60	66	38 (173%)	102	109	87	402	315 (360%)	44	13	31 (-70.45%)
Total for MyCollaboration		**30**	**95**	**318**	**344**	**223 (235%)**	**473**	**547**	**486**	**2018**	**1532 (315%)**	**249**	**100**	**149 (-59.84%)**
Dolibarr	R1-R2	3	20	73	97	53 (265%)	239	254	68	452	383 (560%)	77	39	38 (-49.35%)
	R2-R3	3	23	87	95	64 (278%)	261	270	79	362	284 (361%)	72	17	55 (-76.39%)
	R3-R4	3	19	82	102	63 (332%)	232	255	62	360	298 (480%)	83	43	40 (-48.19%)
	R4-R5	3	21	89	98	68 (324%)	220	243	70	360	290 (414%)	77	32	45 (-58.44%)
	R5-R6	3	20	90	102	70 (350%)	229	243	47	400	353 (748%)	82	26	56 (-68.29%)
	R6-R7	3	25	98	118	73 (292%)	240	265	56	374	318 (573%)	93	45	48 (-51.61%)
	R7-R8	3	17	81	95	64 (376%)	259	267	154	457	303 (196%)	78	22	56 (-71.79%)
Total for Dolibarr		**21**	**145**	**600**	**707**	**455 (314%)**	**1680**	**1797**	**536**	**2765**	**2229 (416%)**	**562**	**224**	**338 (-60.14%)**
YourContacts	R1-R2	13	32	105	135	73 (228%)	201	225	114	536	422 (371%)	103	54	49 (-47.57%)
	R2-R3	6	53	161	201	108 (204%)	250	255	132	631	499 (378%)	148	45	103 (-69.59%)
	R3-R4	5	47	172	182	125 (266%)	345	376	132	728	596 (451%)	135	41	94 (-69.63%)
	R4-R5	23	85	196	233	111 (131%)	341	350	130	636	507 (391%)	148	46	102 (-68.92%)
	R5-R6	8	25	164	188	139 (556%)	356	369	133	538	404 (303%)	163	37	126 (-77.30%)
	R6-R7	3	62	161	173	99 (160%)	280	345	132	531	399 (303%)	111	77	34 (-30.63%)
	R7-R8	2	71	185	213	114 (161%)	321	342	133	428	296 (223%)	142	42	93 (-65.49%)
	R8-R9	8	38	133	146	95 (250%)	302	333	131	636	505 (385%)	108	44	64 (-59.26%)
	R9-R10	8	67	170	186	103 (154%)	328	389	131	538	407 (311%)	119	77	42 (-35.29%)
	R10-R11	4	54	140	161	86 (159%)	309	356	133	542	409 (308%)	107	68	39 (-36.45%)
	R11-R12	8	81	153	171	72 (89%)	241	267	133	625	492 (370 %)	90	44	46 (-51.11%)
Total for YourContacts		**88**	**615**	**1740**	**1989**	**1125 (183%)**	**3274**	**3607**	**1433**	**6367**	**4935 (344%)**	**1374**	**582**	**792 (-57.64%)**
Total across all Web Apps		**389**	**1431**	**4420**	**4948**	**2989 (209%)**	**8034**	**8828**	**5908**	**29679**	**23770 (402%)**	**3517**	**1322**	**2195 (-62.41%)**

tions considered. Column 2 lists the pairs of releases that serve as start/end points for sequences of versions. Column 3 lists the numbers of intermediate versions that occur within these sequences of versions. Columns 4 and 5 report the numbers of coarse-grained and fine-grained repairs performed, respectively, on each sequence of versions by WATER and WATERFALL. Column 6 reports the number of total repairs needed for each sequence of versions. Column 7 reports the differences between the number of fine-grained repairs performed and the number of coarse-grained repairs performed, along with the percentage increase in repairs performed. Column 8 reports the number of repairs made by WATERFALL for all sequences of intermediate versions lying between releases R

and R'. Column 9 reports the number of repairs needed for all sequences of intermediate versions lying between releases R and R'. We defer discussion of the other columns to Sections 5.5.2 and 6. Rows shaded in gray provide totals across all sequences of versions per individual web application, and (bottom) all web applications.

Across all web applications, considering overall totals, the fine-grained repair approach was able to repair 4420 out of 4948 (89.3%) locator breakages, while the coarse-grained approach was able to repair only 1431 of 4948 (28.9%) locator breakages; WATERFALL was thus 209% more effective than WATER overall. Improvements on individual web applications were all relatively large, ranging from 183% on *YourContacts* to 314% on *Dolibarr*.

The results show the improved effectiveness of the fine-grained repair approach applied across all of the sequences considered. For example, consider *YourContacts*, the application in which the widest variation in the numbers of coarse-grained repairs across sequences of versions occurred. Here, the overall improvement of fine-grained repair over coarse-grained repair was 183%. The lowest level of improvement occurred on the sequence of versions $R11$–$R12$ (89%), and the highest level occurred on the sequence of versions $R5$–$R6$ (556%). Aside from these outliers, however, all other improvement percentages fell within a smaller range of 131% to 228%.

To assess whether the observed differences in effectiveness were statistically significant we applied Mann-Whitney tests [15] to the data, on a per-program basis, testing the null hypothesis that the two approaches did not differ in effectiveness at a confidence level of 95%. In all cases the differences were statistically significant (we omit the data due to space limitations).

5.5.2 RQ2: Efficiency

Columns 10 and 11 in Table 4 indicate, for each web application, how many seconds the coarse- and fine-grained approaches spent applying repair techniques. Column 12 calculates the difference in time between the two approaches, and the percentage overhead in time required by the coarse-grained approach with respect to the fine-grained approach. The fine-grained repair approach (WATERFALL) required more time than the coarse-grained approach (WATER), on all applications and sequences of versions. Differences ranged (across sequences of versions) from 439 to 844 seconds on *PHPAddressBook*, from 427 to 666 seconds on *PHPAgenda*, from 311 to 439 seconds on *PHPFusion*, from 288 to 519 seconds on *Joomla*, from 272 to 382 seconds on *MyCollaboration*, from 284 to 383 seconds on *Dolibarr*, and from 296 to 596 seconds on *YourContacts*. On individual sequences of versions, the smallest time difference was 272 seconds (4.55 minutes) on sequence $R4 - R5$ of *MyCollaboration* and the largest was 884 seconds (14.73 minutes) on sequence $R8 - R9$ of *PHPAddressBook*.

We applied Mann-Whitney tests to the data on a per-program basis, testing the null hypothesis that the two approaches did not differ in efficiency at a confidence level of 95%. In all cases the differences were statistically significant (we omit the data due to space limitations).

We now consider the second component of efficiency: the number of test repairs that the coarse- and fine-grained approaches require engineers to accomplish manually. We denote these two costs by C_M and F_M, respectively. They are displayed, for each sequence of versions, in Columns 13 and 14 of the table. Column 15 reports the differences between F_M and C_M, along with the percentage increase in repairs required by C_M.

As the data shows, the fine-grained repair approach required far fewer manual repairs, overall, than the coarse-grained approach. Across all sequences of versions, the differences range from seven (on *PHPAddressBook*, sequence $R8 - R9$, the coarse-grained approach required 19 manual repairs while the fine-grained approach required only 12) to 126 (on *YourContacts* sequence $R5 - R6$, the coarse-grained approach required 163 manual repairs while the fine-grained approach required only 37).

We again applied Mann-Whitney tests to the data on numbers of test repairs, on a per program basis; again, in all cases the differences were statistically significant.

6. DISCUSSION

6.1 Cost-Effectiveness

Comparisons of the execution times of the coarse- and fine-grained approaches favor the former, but considering these in conjunction with our other results suggests a different picture. Our own experience repairing tests (which we were required to do for all tests of the web applications we considered) attests to the costliness of the task. For example, for *PHPAddressBook*, the first author spent approximately 40 hours manually repairing the 155 breakages that occurred in the application's tests across its versions; this amounts to an average cost of over 15 minutes per breakages. In one extreme case, over two hours were required to repair a breakage.

Keeping this in mind, consider the efficiency and effectiveness results obtained on the sequence of versions $R8 - R9$ of *PHPAddressBook*. Here, the difference in the number of manual repairs that must be performed by engineers using the two approaches is at its *lowest*: with coarse-grained repair, 19 (35-16) manual repairs must still be performed by engineers, whereas with fine-grained repair, 12 must be performed (two (35-33) on R' and 10 (58-48) on intermediate versions). In this case, coarse-grained repair requires 142 seconds, and fine-grained repair requires 986 seconds – just over 14 minutes more. Thus, the coarse-grained approach is more *cost-effective* than the fine-grained approach only if manual repairs of the seven additional unrepaired breakages can be accomplished in less than 14 minutes (an average of 44 seconds per breakage in our example). Given our experiences with manual breakage repair it seems unlikely that this would be the case in practice.

Next, consider the sequence of versions $R5 - R6$ on *YourContacts*. Here, the difference in the number of repairs that must be performed by engineers using the two approaches is at its *highest*: with coarse-grained repair, 163 (188-25) repairs must be performed by engineers, whereas with fine-grained repair, 37 must be performed (24 (188-164) on R' and 13 (369-356) on intermediate versions). In this case, coarse-grained repair requires 133 seconds, and fine-grained repair requires 538 seconds – almost seven minutes more. In this case, the coarse-grained approach is more *cost-effective* than the fine-grained approach only if manual repair of the 126 additional unrepaired breakages can be accomplished in less than seven minutes (an average of 3.33 seconds per breakage in our example). This seems entirely impossible.

Based on our own experiences manually repairing test breakages, it also seems possible that it may be easier to repair breakages in intermediate versions than in the final release of a sequence. This is because in intermediate versions, web applications have undergone fewer, smaller changes with respect to prior versions, rendering breakages easier to analyze and repairs easier to conduct. On the other hand, there might be some startup costs associated with repairs that can be amortized when considering multiple breakages at once, rendering human effort in the coarse-grained approach less expensive. Empirical studies that measure human costs associated with repairs will be needed to investigate these scenarios.

6.2 Repairing Versions Versus Commits

The first five web applications listed in our tables were considered at the level of releases and intermediate versions, whereas the last two (*Dolibarr* and *YourContacts*) were considered at the level of releases and commits. Where intermediate repairs completed and needed are concerned, *Dolibarr* and *YourContacts* have higher numbers than most of the other applications; however, these numbers are similar to those obtained on *PHPFusion*. Investigating this further, we found that this high number of manual repairs was needed on these applications because they contained a substantially higher number of non-locator breakages than the other four web applications. Thus, the differences in numbers of manual repairs was not due to the use of commits.

6.3 The Potential for Backtracking

To obtain some initial insights into the use of an enhanced version of WATERFALL incorporating backtracking (Section 4.3), we simulated the use of such an algorithm by considering the sequence of versions for which WATERFALL was the least effective at performing repairs: $R2 - R3$ of *YourContacts*. In this case, the fine-grained repair approach failed to repair 40 of 201 breakages. The first author simulated the backtracking version of WATERFALL by considering the subset of tests that WATERFALL was not able to repair for sequence of versions $R2 - R3$ of *YourContacts*. *YourContacts* has 57 tests, of which WATERFALL repaired 33 initially, leaving 24 unrepaired. The simulation involved capturing all of the sequences of suggested repairs made by WATERFALL for any breakage and then backtracking from R' to R and running the test with these suggested sequences of repairs. We simulated the ongoing applications of WATERFALL to all sequences of suggested repairs, considering all possible combinations as the backtracking approach might do if applied exhaustively. The process was halted whenever a test was repaired across one of these sequences. A test was considered to be repaired if it was found to run properly through the last version of the sequence. We repeated this process for all 24 unrepaired tests in the sequence of versions.

Using this simulated backtracking approach, we were able to find repairs for 33 of the 40 breakages that were not addressed by our basic algorithm. While this simulation does not take into account efficiency, keep in mind that it did reduce the number of breakages that would need manual attention by 33 (+82.5% of the repairs with respect to the basic algorithm), and thus is worth a certain amount of overhead. Assessing whether the cost-effectiveness tradeoff of a backtracking algorithm would be worthwhile in practice, however, requires further study.

7. RELATED WORK

There has been some research on automated repair of *programs* (e.g., [4, 10, 20, 24, 33–35]). Our work focuses on tests.

There have been numerous papers on test repair. Several papers have addressed the problem of repairing unit tests such as JUnit tests or tests written in similar frameworks (e.g., [6, 7, 23]).

Many researchers have attempted to repair GUI tests, which track sequences of user actions applied to an interface. Memon and Soffa [22], Memon [21] and Datchayani et al. [9] use event flow graphs and transformation techniques to repair broken GUI tests. Huang et al. [14] use a genetic algorithm to repair GUI tests. Grechanik et al. [11] analyze an initial and modified GUI for differences and generate a report for engineers documenting ways in which test scripts may be broken by changes. Daniel et al. [8] use GUI refactorings to track the changes engineers make to a GUI, suggesting that this information could be used to repair tests. Such approaches, however, do not address directly web tests produced by record/replay tools, although they might be adapted to do so.

Zhang et al. [38] address the problem of repairing broken *workflows* in GUI applications, where a workflow is a sequence of activities to perform a given task. This approach, however, does not attempt to repair actual tests.

Alshawan and Harman [1] present an approach for repairing user session data that can be collected and used to regression test web applications. User session data, however, differs from the data captured by record/replay tools, because it includes only requests received by a server from the web application.

Other research attempts to simplify the debugging of faults in web applications. Hammoudi et al. [12] use delta debugging to reduce recordings that expose failures to smaller versions. Wang et al. [32] use dynamic slicing to discard events from an execution trace that are not required for failure reproduction. These approaches might be used to facilitate test repair by reducing tests to a subset of actions that still result in breakages.

Recent papers consider problems related to the robustness and maintainability of web test suites. Stocco et al. [30, 31] investigate the automated generation of page objects that confine causes of test breakages to a single class, a form of breakage prevention. Yandrapally et al. [36] address the problem of test script fragility in relation to locators, proposing approaches for robustly identifying UI elements by using contextual clues, which is also a form of prevention. Another approach for producing robust locators has been implemented in the tools ROBULA [16] and ROBULA+ [18]. Bajaj et al. [2, 3] present LED, a tool that automatically synthesizes web element locators by solving a constraint satisfaction problem over the group of valid DOM states in a web application.

To our knowledge, there have been only two papers that have presented techniques for repairing locator breakages in web tests. We have already discussed the technique proposed by Choudhary et al. [5]. In addition, the multi-locator extension of ROBULA [17] supports automated repair of broken locators by identifying and attempting to apply other potential locators generated by different tools at the breakage site. WATERFALL, in contrast, is based on differential testing.

8. CONCLUSIONS AND FUTURE WORK

We have presented WATERFALL, a fine-grained approach for repairing record/replay tests of web applications. We have conducted an empirical study comparing our approach to a coarse-grained approach on seven non-trivial open-source web applications. Our results show that WATERFALL is far more effective than the coarse-grained approach at automatically repairing tests. Moreover, while WATERFALL requires more execution time than the coarse-grained approach, the extra overhead is overshadowed by the reduction in manual test repairs it requires.

As future work, we intend to extend our implementation of WATERFALL to handle other classes of breakages that are not necessarily confined to locators (e.g., repairing assertions), with the goal of repeating the study both at coarse- and fine-grained levels. We also intend to develop the backtracking version of WATERFALL, in order to compare it with the multi-locator repair approach suggested by Leotta et al. [17]. Finally, since our approach has demonstrated promise, we intend to conduct an empirical study of humans to assess its cost-effectiveness in practice.

9. ACKNOWLEDGEMENTS

This work has been partially supported by the National Science Foundation through award IIS-1314365.

10. REFERENCES

[1] N. Alshawan and M. Harman. Automated session data repair for web application regression testing. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*, pages 298–307, 2008.

[2] K. Bajaj, K. Pattabiraman, and A. Mesbah. Led: Tool for synthesizing web element locators. In *Proceedings of the International Conference on Automated Software Engineering*, pages 848–851, 2015.

[3] K. Bajaj, K. Pattabiraman, and A. Mesbah. Synthesizing web element locators. In *Proceedings of the International Conference on Automated Software Engineering*, pages 331–341, 2015.

[4] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *Proceedings of the International Conference on Software Engineering*, pages 121–130, 2011.

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

[6] B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 207–218, 2010.

[7] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. ReAssert: Suggesting repairs for broken unit tests. In *Proceedings of the International Conference on Automated Software Engineering*, pages 433–444, 2009.

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

[9] M. Dhatchayani, X. A. R. Arockia, P. Yogesh, and B. Zacharias. Test case generation and reusing test cases for GUI designed with HTML. *Journal of Software*, 7(10):2269–2277, 2012.

[10] D. Gopinath, M. Z. Malik, and S. Khurshid. Specification-based program repair using SAT. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–188, 2011.

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

[12] M. Hammoudi, B. Burg, G. Bae, and G. Rothermel. On the use of delta debugging to reduce recordings and facilitate debugging of web applications. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 333–344, 2015.

[13] M. Hammoudi, G. Rothermel, and P. Tonella. Why do record/replay tests of web applications break? In *Proceedings of the International Conference on Software Testing*, 2016.

[14] S. Huang, M. B. Cohen, and A. M. Memon. Repairing GUI test suites using a genetic algorithm. In *Proceedings of the International Conference on Software Testing*, pages 245–254, 2010.

[15] O. Koresteleva. *Nonparametric Methods in Statistics with SAS Applications*. CRC Press, Boca Raton, FL, 2004.

[16] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Reducing web test cases aging by means of robust XPath locators. In *Proceedings of the International Symposium on Software Reliability Engineering Workshops*, pages 449–454, 2014.

[17] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Using multi-locators to increase the robustness of web test cases. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 1–10, 2015.

[18] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. ROBULA+: An algorithm for generating robust XPath locators for web testing. *Journal of Software: Evolution and Process*, 28(3):177–204, 2016.

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

[20] S. Mechtaev, J. Yi, and A. Roychoudhury. DirectFix: Looking for simple program repairs. In *Proceedings of the International Conference on Software Engineering*, pages 448–458, 2015.

[21] A. M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Transactions on Software Engineering and Methodology*, 18(2):4:1–4:36, 2008.

[22] A. M. Memon and M. L. Soffa. Regression testing of GUIs. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2003.

[23] M. Mirzaaghaei, F. Pastore, and M. Pezze. Supporting test suite evolution through test case adaptation. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 231–240, 2012.

[24] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the International Conference on Software Engineering*, pages 772–781, 2013.

[25] S. Raemaekers, A. van Deursen, and J. Visser. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software*, 2016 (to appear).

[26] Sahi. sahipro.com.

[27] The Selenium Project. http://seleniumhq.org/docs/03_webdriver.html/.

[28] Watir WebDriver. http://watirwebdriver.com.

[29] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 488–498, 2013.

[30] A. Stocco, M. Leotta, F. Ricca, and P. Tonella. Why creating web page objects manually if it can be done automatically? In *Proceedings of the International Workshop on Automation of Software Test*, pages 70–74, 2015.

[31] A. Stocco, M. Leotta, F. Ricca, and P. Tonella. Clustering-aided page object generation for web testing. In *Proceedings of the International Conference on Web Engineering*, pages 132–151, 2016.

[32] J. Wang, W. Dou, C. Gao, and J. Wei. Fast reproducing web application errors. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 530–540, 2015.

[33] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buckholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the International Symposium on Software Testing*, pages 61–72, 2010.

[34] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen.

Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116, 2010.

[35] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the International Conference on Software Engineering*, pages 364–374, 2009.

[36] R. Yandrapally, S. Thummalapenta, S. Sinha, and S. Chandra. Robust test automation using contextual clues. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 304–314, 2014.

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

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