Software Quality Journal (2022) 30:1089–1121
https://doi.org/10.1007/s11219-022-09596-z

SleepReplacer: a novel tool-based approach for replacing thread sleeps in selenium WebDriver test code

Dario Olianas[1] · **Maurizio Leotta**[1] · **Filippo Ricca**[1]

Accepted: 28 June 2022 / Published online: 12 August 2022

Abstract
Assuring quality of web applications is fundamental, given their relevance in the today's world. A possible way to reach this goal is through end-to-end (E2E) testing, an approach in which a web application is automatically tested by performing the actions that a user would do. With modern web applications (for example, single-page applications), it is of great importance to properly handle asynchronous calls in the test suite. In E2E Selenium WebDriver test suites, asynchronous calls are usually managed in two ways: using thread sleeps or explicit waits. The first is easier to use, but is inefficient and can lead to instability (also called flakiness, a problem often present in test suites that makes us lose confidence in the testing phase), while the second is usually more efficient but harder to use because, if the correct kind of wait is not carefully selected, it can introduce flakiness too. To help Testers, who often opt for the first strategy, we present in this work a tool-based approach to automatically replace thread sleeps with explicit waits in an E2E Selenium WebDriver test suite without introducing new flakiness. We empirically validated our tool named SleepReplacer on four different test suites, and we found that it can correctly replace in an automatic way from 81 to 100% of thread sleeps, leading to a significant reduction of the total execution time of the test suite (i.e., from 13 to 71%).

Keywords Web testing · Thread sleeps · Explicit waits · Test code · Selenium WebDriver

1 Introduction

Regression testing is playing an increasingly important role in the industrial context (Ali et al., 2019), in particular in software development processes as DevOps (Zhu et al., 2016) where continuous integration, continuous testing, and continuous delivery are adopted. This is mainly due because regression testing, if well conducted, ensures that changes to the system under test during software evolution do not break existing functionality.

✉ Maurizio Leotta
maurizio.leotta@unige.it

[1] Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), University of Genova, Genova, Italy

However, for regression testing to be cost-effective, the test suite must be efficient (Ekelund & Engström, 2015; Hossain et al., 2014) and reliable. This is particularly true in the Web environment (Eda & Do, 2019) where testing is conducted not only at the unit and integration level, but also at the system level applying E2E testing frameworks such as Selenium WebDriver (SeleniumHQ, 2021; García et al., 2020). In fact, in this context the Testers often execute a very large number of automated test cases (implemented as E2E test methods) since Web applications are often very complex and modified frequently (Eda & Do, 2019).

Delays due to inefficient test suites and flaky tests are two big problems, often coupled, that can cause cost increases. Concerning the first aspect, the cost is twofold: both in the time that Testers spend waiting for tests to finish running and in the monetary cost of running tests on computers. Instead, flaky tests are even more insidious and dangerous (Luo et al., 2014). The fact that a test can non-deterministically pass or fail when executed on the same version of the Application Under Test (AUT), without any change in both the app and the test code, can waste a lot of time for Testers trying to debug a non-existent fault in the code (Lam et al., 2020).

The two aspects above mentioned are inherently interrelated because most of the proposed methods to deal with flakiness rely on multiple test repetitions, i.e., a test method is executed against the same version of the AUT for a given number of times, and if it produces different results the test is considered flaky. In this way, potentially inefficient test code can be run multiple times to verify the absence of flakiness thus increasing the overall execution time of the test suite and associated cost.

Often these two problems, in the E2E Web Testing context, derive from a common cause: a bad practice often used by Web Testers (developing their test methods using for instance Selenium WebDriver, one of the testing frameworks most used in this context), namely the use of *thread sleeps* (Ricca & Stocco, 2021). Thread sleeps are commands that stop the execution of the thread for a given amount of time and are used by Testers, at certain specific points in test code, to wait for a page of the AUT to load before taking the next action or for managing asynchronous calls, often used in modern Web applications. The usage of *thread sleeps* presents, however, two main disadvantages: (a) they lengthen the execution times of the test code since they always wait the same amount of time, given a priori by the Tester, even when the page is loaded more quickly, and (b) they can cause flakiness themselves, as testified for example in Luo et al. (2014) and Ricca and Stocco (2021) papers. From several years, the Selenium testing framework is offering a better solution to synchronize test code and AUT, called `explicit waits` (García et al., 2020). Explicit waits are more efficient than thread sleeps, because they automatically stop waiting when the expected condition is verified, instead of waiting the fixed amount of time defined by the Tester during test development. On large test suites requiring an extensive use of waits, adopting explicit waits instead of thread sleeps can lead to great time savings. Explicit waits are also more flexible and reliable than thread sleeps, because they allow to check for complex conditions and if used well they can also drastically limit the problem of flakiness.

In this paper, we present a novel tool-based approach named SleepReplacer able to automatically replace thread sleeps with explicit waits in a Selenium WebDriver test suite adopting (or not) the Page Object (PO) pattern (Page Object Model) without introducing novel flakiness and thus saving Tester's precious time and reducing costs.

To the best of our knowledge, this is the first tool in literature capable of performing this task. Carrying out this replacement in automatic way is not trivial because the Selenium framework provides several explicit waits and to select the suitable one is necessary to take into account the type of interaction performed by the test code after the thread sleep to be replaced. The results obtained during the evaluation of SleepReplacer, conducted using a real industrial PO-based test suite (used also for evaluation our previous manual approach (Olianas et al., 2021)) and three smaller test suites for open-source Web applications, are very satisfactory.

This paper was born as an extension of the QUATIC 2021 conference paper (Olianas et al., 2021) even if it has a different goal. The goal of the previous paper was to provide a procedure to remove flakiness, instead the goal of this paper is to present a tool's based approach able to automatically refactor a test suite, removing as much thread sleeps as possible. In particular, this paper provides the following novelties: (1) a tool that automatically refactors, without introducing novel flakiness, a test suite using thread sleeps with an equivalent, instead, using explicit waits; (2) three novel test suites employed to evaluate our approach which are added to the industrial one already presented and used in our previous work.

This paper is organized as follows: Sect. 2 introduces thread sleeps and explicit waits and then explains how to do the replacement by hand. Section 3 sketches the tool-based approach we devised to automatically replace thread sleeps with explicit waits. Section 4 adds some implementation details to the previous section. Section 5 presents the research questions of the experimental study and shows the obtained results. Finally, Sect. 6 summarizes related works and Sect. 7 concludes the paper.

2 Replacing Thread.sleep() with explicit waits

In E2E web testing the simpler mechanism to wait for asynchronous calls is the use of thread sleeps. Since thread sleeps are not an optimal solution, because they are inefficient and can become a cause of flakiness themselves (Ahmad et al., 2019; Camara et al., 2021; Shukla, 2021), a Tester may want to replace them with more reliable waiting mechanisms. The Selenium WebDriver framework offers three ways for waiting the loading of web elements: *explicit waits*, *implicit waits* and *fluent waits*.

2.1 Explicit waits

From a technical point of view, an explicit wait is a Java object of class `WebDriverWait`, that can be used in combination with an `ExpectedCondition`, that is a function that tells the explicit wait which condition should be checked to stop waiting. From the Selenium documentation[1] we can identify six main categories of expected conditions that check for different conditions, such as:

- the visibility of an element;
- the clickability of an element;
- the presence of an element;
- the number of elements;
- the page's URL;

[1] Documentation for the ExpectedConditions class. https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/support/ui/ExpectedConditions.html

– text comparison;
– DOM attributes.

Not only explicit waits are more efficient than thread sleeps, they are also more flexible, because explicit waits allow to check for complex conditions: for example, if we have a text in a page that can be dynamically changed via AJAX, and we want to wait for the text to assume a certain value, it would be difficult to do it using thread sleeps. But with the `textToBe` expected condition, this check requires just a single line of code.

Listing 1 shows an example of thread sleep from one of the test suites used to validate our tool, while Listing 2 shows the code produced by our tool to replace that thread sleep.

```
driver.findElement(By.id("add_butn")).click();
Thread.sleep(1000);
driver.findElement(By.id("name")).clear();
driver.findElement(By.id("name")).sendKeys("Milestone001");
```

Listing 1. Example of thread sleep

```
WebDriverWait wait = wait = new WebDriverWait(driver, 10);
...
driver.findElement(By.id("add_butn")).click();
wait.until(ExpectedConditions.elementToBeClickable(By.id("name")));
driver.findElement(By.id("name")).clear();
driver.findElement(By.id("name")).sendKeys("Milestone001");
```

Listing 2. Example of explicit wait

To replace a thread sleep with an explicit wait, we need some information about (a) the condition that we need to wait for (b) the web page element involved. This information can usually be inferred from the test method code, although in some complex cases a page inspection may be required. We can infer which condition should be waited by looking at the web page interaction performed after the thread sleep: actions like clicks and writing text in fields can be waited with the `elementToBeClickable` expected condition; actions that read text from an element usually require that the element is visible, and so the expected condition `visibilityOf` can be used. There are many other expected conditions to deal with JavaScript alerts (`alertIsPresent`), frames (`frameToBeAvailableAndSwitchToIt`), text that can be changed dynamically (`textToBe`), selection state of an element (`elementToBeSelected`) and many others.

When a WebDriverWait object is created (e.g., first line in Listing 2), the Tester must specify the waiting timeout. If the expected condition did not happen before the timeout is passed, a `TimeoutException` will be thrown, making the test to fail. As previously said, the WebDriverWait object stops waiting when the ExpectedCondition is verified, so it is a good practice to set a timeout much larger than the usual waiting time: in this way, if some transient network problem arises and an action requires more time than usual, the test method will not fail. Moreover, since the waiting stops when the condition is verified, increasing the timeout will not affect the execution time of the test suite.

2.2 Implicit waits

Implicit waits are another waiting mechanism offered by the Selenium WebDriver framework. An implicit wait is set globally to the WebDriver object, and makes it wait a specific amount of time before every interaction with the web page. The code that sets an implicit wait of 5 s is shown in Listing 3.

```
driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);
```

Listing 3. Example of implicit wait setting

When an implicit wait of 5 s is set, and an interaction with a page element is made, the WebDriver will poll the DOM for 5 s waiting for the page element to appear. This behavior may look similar to the explicit waits, but there are some important differences, well explained for instance in Chapter 10 of the book "Python Testing with Selenium" (Raghavendra, 2021). One of the most important is that implicit waits are set globally, while explicit waits locally: by using implicit waits we will use the same timeout for all interactions, while with explicit waits it can be changed for each interaction. Moreover, even more important is that implicit waits can only wait for the presence of an element, while explicit waits can wait for many different ExpectedConditions. A Tester may think to use an implicit wait globally, and integrate it with explicit waits when required, but this is discouraged in the Selenium documentation[2]: in fact, mixing implicit and explicit waits can lead to unpredictable, potentially infinite waiting times (Raghavendra, 2021). That is why many academic and professional sources recommend to use explicit waits as best practice in web testing.

2.3 Fluent waits

Finally, fluent waits are a more customizable version of the explicit waits. In the Selenium Java implementation, FluentWait is the superclass of WebDriverWait: the underlying polling mechanism is the same, but FluentWaits allow the Tester to customize more parameters than the WebDriverWait class (the one used for explicit waits) such as polling frequency, ignored exceptions and error message to be displayed when the timeout expires. Listing 4 shows the code required to use a FluentWait.

```
Wait<WebDriver> wait = new FluentWait<WebDriver>(driver)
  .withTimeout(Duration.ofSeconds(30))
  .pollingEvery(Duration.ofSeconds(5))
  .ignoring(NoSuchElementException.class);

wait.until(ExpectedConditions.elementToBeClickable(By.id("name")));
```

Listing 4. Example of fluent wait

[2] Selenium WebDriver documentation for waits https://www.selenium.dev/documentation/webdriver/waits/

In our work, we preferred explicit waits over fluent waits mainly because (1) explicit waits are well known and more used in the Web testing community, and (2) to fully exploit the potential of fluent waits, it is necessary to tune more parameters (e.g., polling time, ignored exceptions) than with explicit waits. On the other hand, using fluent waits with default polling time and no ignored exceptions is equivalent to use explicit waits.

2.4 Page object pattern

Both thread sleeps and explicit waits can be inserted directly in the test code or, alternatively, in the Page Objects classes (Page Object Model). Since Page Objects can be often found in high-quality industrial test suites (Leotta et al., 2013, 2020), we devised our approach in order to be able to replace thread sleeps commands also in such classes and not only in plain test code. In the following of the section, we briefly describe the Page Object pattern (Page Object Model), a design pattern that aims to reduce code duplication and improve test maintenance. It is an object-oriented design pattern, where there are classes that act like interfaces to the pages of the web application under test. For example, let's assume we have a simple login page with a form that contains a text box for the username, a text box for the password and a Login button to submit the form and we want to test it. A plain test code for the login page is given in Listing 5 (where we assume that the driver object has already been initialized).

```
public class TestClass {
...
@Test
public void loginTest() {
    driver.get("http://www.example.com/login");
    driver.findElement(By.id("username")).sendKeys("yourUsername");
    driver.findElement(By.id("password")).sendKeys("yourPassword");
    driver.findElement(By.id("loginBtn")).click();
    assertThat(driver.findElement(By.id("loginResult")).getText(), is(
        "login successfull!");
}
...
}
```

Listing 5. Plain test code (i.e., without page objects) for our login page

This test code has several problems, and the biggest one is that locators (e.g., By.id("username")) are hardwired in the test code and duplicated among the test methods: if a locator changes due to web application evolution, we have to manually change it in every point where it is used (i.e., in every test method). By using the Page Object pattern, instead, we encapsulate all the code that interacts with the page in a unique LoginPage class and we can reuse it. The refactored test method using the Page Object pattern is shown in Listing 6.

```
public class LoginPO {

    private WebDriver driver;

    ...

    public LoginPO goToLoginPage() {
        driver.get("http://www.example.com/login");
        return this;
    }

    public LoginPO login(String username, String password) {
        driver.findElement(By.id("username")).sendKeys(username);
        driver.findElement(By.id("password")).sendKeys(password);
        driver.findElement(By.id("loginBtn")).click();
        return this;
    }

    public boolean getLoginStatus() {
        return driver.findElement(By.id("loginResult")).getText().
            equals("Login successfull!");
    }

}

public class TestClass {
...
    @Test
    public void loginTest() {
        boolean success = new LoginPO(driver)
            .goToLoginPage()
            .login("yourUsername", "yourPassword")
            .getLoginStatus();
        assertTrue(success);

    }
...
}
```

Listing 6. Test code with page objects for our login page

In this way the test code is much more compact, and moreover the locators are encapsulated in the page object classes, making test suite maintenance easier.

3 Our tool-based approach

Our approach aims to remove thread sleeps in a Selenium WebDriver test suite and to replace them with explicit waits when possible. The main goal is to reduce the execution time of the entire test suite without introducing novel flakiness. To avoid introducing or augmenting flakiness, our approach prescribes to proceed step by step, validating each change applied to the code immediately, by running the modified test method for a given amount of times and observing that the change did not introduce side-effects. This mechanism is conservative, because in this way, each time a validation fails, we know that the cause of failure can only be the last change introduced in the code. Because of this conservative mechanism, the test suite in input must not be flaky. In fact, having flakiness in the original test suite makes identification of the root cause of failures harder: if the original test suite is not flaky, every time we find a flakiness behavior we

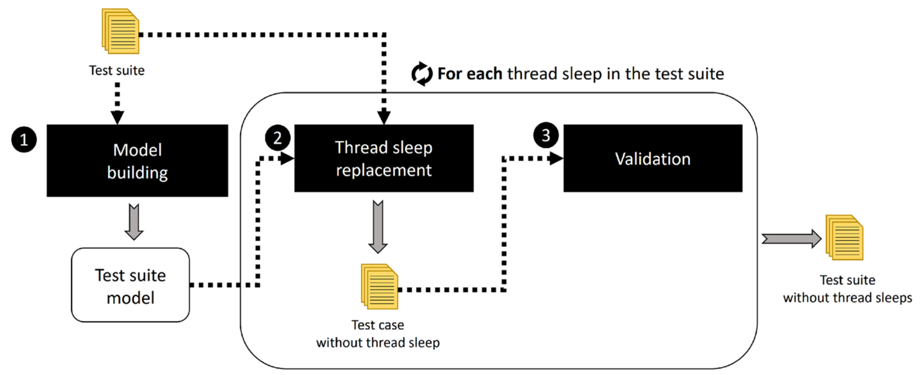

Test suite
For each thread sleep in the test suite
1
Model building
2
Thread sleep replacement
3
Validation
Test suite model
Test case without thread sleep
Test suite without thread sleeps

Fig. 1 Workflow of our approach

are sure that is due to the last change made to the code, but if the initial test code can fail non-deterministically this assumption is no longer valid.

Unfortunately, our approach is not able to remove existing flakiness in test suites. Removing flakiness is a very hard task, which strictly depends on the characteristics of the test suite and the application under test, and for which is not easy to give general guidelines. However, some approaches to identify the root cause of flakiness (Moran et al., 2020) or fixing flaky tests exist in literature (Shi et al., 2019): the first one is oriented to end-to-end web testing, while the second one is limited to resolving flakiness caused by order-dependent tests. If, instead, the test suite is affected by a small amount of flakiness (e.g., it shows only in few tests, always in the same points), the Tester can try to fix it by adding a thread sleep where required, that will be lately replaced with an explicit wait by the tool implementing our approach.

We thought our approach to be capable of working whether the test suite is designed with the PO pattern or without. We have decided to manage both possibilities (PO yes and PO no) to increase the usage scenarios of our approach. As described before, a page object is an object-oriented class that serves as an interface toward a Web page of the application under test (Page Object Model). Test methods use the methods offered by PO classes whenever they need to interact with an element of the web app user interface.

The workflow of our approach is represented in Fig. 1 and can be summarized as follow:

1. **build the model** of the test suite
2. **for each** thread sleep in the test suite:

    (a) **replace** the thread sleep with an explicit wait with the appropriate expected condition (or remove it)
    (b) **validate** the modified code: if it is OK move to the next thread sleep, otherwise before moving restore the removed thread sleep at the previous step

3.1 Step 1 — model building phase

Algorithm 1: BuildModel procedure for building the model of the test suite

```
    Input  : TSuite – a test suite with thread sleeps
    Output : $TSuite_M$ – a model of the test suite
1   BuildModel(TSuite):
2       $TSuite_M \leftarrow$ new $TestSuite_M()$ // creates a new model instance representing TSuite
3       foreach test class TClass in TSuite do
4           $TClass_M \leftarrow$ new $TestClass_M()$ // creates a new model instance representing TClass
5           foreach test method TMethod in TClass do
6               $TMethod_M \leftarrow$ new $TestMethod_M()$ // creates a new model instance representing TMethod
7               add $TMethod_M$ to $TClass_M$
8           add $TClass_M$ to $TSuite_M$
9       // at this point the model $TSuite_M$ represents the structure of the entire test suite in terms of a hierarchy of
          $TClass_M$ and $TMethod_M$ but without information on the thread sleeps (TSleep) and page accesses (PA).
          Such info are added in the next steps depending on whether the test suite TSuite adopts or not the Page
          Object pattern
10      // Case TSuite is PO-based
11      if TSuite is based on the Page Object pattern then
12          foreach page object PO in TSuite do
13              $PO_M \leftarrow$ new $PageObject_M()$ // creates a new model instance representing PO
14              foreach method POMethod in PO do
15                  $POMethod_M \leftarrow$ new $PageObjectMethod_M()$ // creates a new model instance representing
                      POMethod
16                  usages ← all test methods that call POMethod
17                  add usages to $POMethod_M$
18                  foreach thread sleep TSleep in a line L of POMethod do
19                      $TSleep_M \leftarrow$ new $ThreadSleep_M()$ // creates a new model instance representing
                          TSleep
20                      $TSleep_M$.location ← L
21                      if there is a page access PA of type PAType at line PALine after TSleep then
22                          $TSleep_M$.pageAccess ← new $PageAccess_M$(PALine, PAType);
23                      add $TSleep_M$ to $POMethod_M$
24                  add $POMethod_M$ to $PageObject_M$
25              add $PageObject_M$ to $TSuite_M$
26      // Case TSuite is not PO-based
27      else
28          foreach test class TClass in TSuite do
29              foreach test method TMethod in TClass do
30                  find $TMethod_M$ corresponding to TMethod in $TSuite_M$
31                  foreach thread sleep TSleep in a line L of TMethod do
32                      $TSleep_M \leftarrow$ new $ThreadSleep_M()$ // creates a new model instance representing
                          TSleep
33                      $TSleep_M$.location ← L
34                      if there is a page access PA of type PAType at line PALine after TSleep then
35                          $TSleep_M$.pageAccess ← new $PageAccess_M$(PALine, PAType);
36                      add $TSleep_M$ to $TMethod_M$
```

Step 1 is required to enable automated interaction with the test suite: in fact, in order to perform its tasks, our tool must know where thread sleeps and web page interaction commands (i.e., page accesses) are located in the test code. Thanks to the model, the tool implementing our approach can know where the thread sleeps are located and which type of page access is performed after them. Indeed, to correctly replace a thread sleep with an explicit wait it is necessary to know which expected condition use and which page element to wait: our tool performs the replacement relying on the information stored in the model. The procedure that implements Step 1 is described in Algorithm 1. The structure of the model depends on the adoption of the PO pattern (or not) by the test suite. If the test suite employs the PO pattern, the model must represent the location of thread sleeps and page accesses in page objects, and the use of PO methods in test methods. Otherwise, if the test suite does not adopt the PO pattern, the model must only represent the location of thread sleeps and page accesses directly in test methods. The class diagram representing the structure of the instance (i.e., meta-model) generated by our tool of the test suite model is represented in Fig. 2. The {and} relationships in the class diagram mean that if an entity exists, also the other must exist; the {xor} relationships mean that if an entity exists, the other must not exist. In this way, we can have a single diagram for both PO-based and non-PO-based test suites.

3.2 Step 2.(a) — thread sleep replacement phase

Algorithm 2: ReplaceSleep procedure for replacing a thread sleep with a explicit wait

```
Input  : TSuite – a test suite with thread sleeps
         TSuite_M – a model of the test suite
         {R} – replacement rules
         iterations – number of validation runs
Output : TSuiteNew – the test suite T with explicit waits in place of thread sleeps
ReplaceSleep (TSuite, TSuite_M, {R}, iterations) :
    if the test suite TSuite has page objects then
        foreach page object PO in TSuite do
            foreach method POMethod in PO do
                foreach thread sleep TSleep in a line L of POMethod do
                    find TSleep_M corresponding to TSleep in TSuite_M
                    // TSleep_M.pageAccess is null if there is no page access after TSleep or if we have no
                      rule to replace the page access after TSleep
                    if TSleep_M.pageAccess is not null then
                        apply the correct rule from {R} to replace TSleep with an explicit wait
                    else
                        remove TSleep
                    if validate(POMethod, TSuite_M, iterations) then
                        continue
                    else
                        remove the last inserted explicit wait (if any) and restore TSleep
    else
        foreach test class TClass in TSuite do
            foreach test method TMethod in TClass do
                foreach thread sleep TSleep in a line L of TMethod do
                    find TSleep_M corresponding to TSleep in TSuite_M
                    // TSleep_M.pageAccess is null if there is no page access after TSleep or if we have no
                      rule to replace the page access after TSleep
                    if TSleep_M.pageAccess is not null then
                        apply the correct rule from {R} to replace TSleep with an explicit wait
                    else
                        remove TSleep
                    if validate(TMethod, TSuite_M, iterations) then
                        continue
                    else
                        remove the last inserted explicit wait (if any) and restore TSleep
```

Step 2 is the fundamental one of our approach and is described in Algorithm 2. The *Step 2.(a)* is performed by looking at the type of access to the web page made after the current thread sleep. In our context, a page access is a Selenium WebDriver command that reads information from the page (e.g., getText(), getAttribute()) or actively interacts with it (e.g., click(), sendKeys()). The

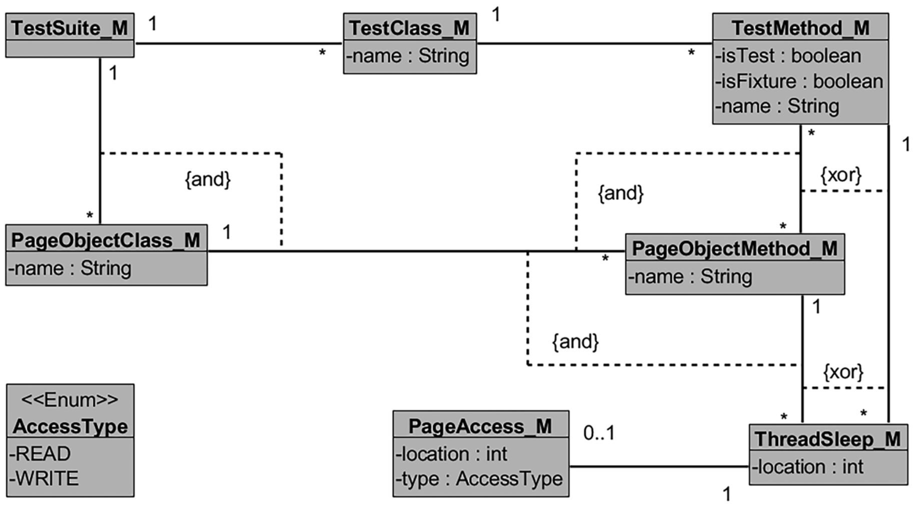


Fig. 2 Test suite meta-model. Our tool generates, for each test suite taken in input, a model complaint with this class diagram

choice of the expected condition, in fact, strictly depends on the type of page access that is made. For example, accesses that only read information from the page (accessType = READ in Fig. 2), without actively interacting with it, usually require an expected condition to wait for an element to be visible. Instead, interactions with the page such as clicks or writing text to a field (accessType = WRITE in Fig. 2), usually require an expected condition that waits for an element to be clickable.

To decide which expected condition should be used depending on the page access that it will protect, our approach relies on a heuristic expressed by means of a set of *replacement rules*. A replacement rule is a function $R : \{A_1, A_2, ..., A_n\} \rightarrow EC$ that maps a set of accesses A to an expected condition EC.

3.3 Step 2.(b) — validation phase

Algorithm 3: Validate subprocedure for validating a novel inserted explicit wait

```
Input  : Method – a test method or page object method
         $TSuite_M$ – a model of the test suite
         iterations – number of validation runs
Output : res – boolean result of validation
1  Validate (Method, $TSuite_M$, iterations) :
2      validationSet ← { }
3      if Method is a page object method POMethod then
4          validationSet ← all test methods that call POMethod by analyzing $TSuite_M$
5      else if Method is a test fixture TFixture then
6          validationSet ← the first test method in the TClass of TFixture by analyzing $TSuite_M$
7      else if Method is a test method TMethod then
8          validationSet ← TMethod by analyzing $TSuite_M$
9      for $i = 0; i < iterations; i{+}{+}$ do
10         if the test suite TSuite has dependencies then
11             (1) load the state required by method Method to run correctly or (2) execute the warranted path
               of the test(s) that are in the validationSet (if no state is available)
12         result ← run the validationSet if result == failure then
13             return false
14     return true
```

Finally, the *Step 2.(b)* requires to run the refactored code for a given amount of executions 'X', decided by the Tester, to be sure that the change did not break the test or introduce novel flakiness. The procedure that implements this step is described in Algorithm 3. To validate a replacement, multiple runs of the modified test may be necessary since, given the non-deterministic nature of flakiness, a single run may not be sufficient to verify if the test is flaky (Palomba, 2019). However, the number of validation runs has a relevant impact on the approach execution time: a high number of runs can heavily increase the execution time of the implementing tool, especially if the original test suite is large and employs the Page Object pattern (see below). On the contrary, using an insufficient number of validation runs may introduce novel flakiness during the refactoring, therefore the decision of this parameter is critical. The Testers should decide the number of validation runs according to their experience and to the history of stability of the test suite: if the test suite already manifested flakiness in the past, a higher number of validation runs may be appropriate.

3.3.1 Effect of test suite structure on the validation process

The way in which the validation is performed strictly depends on how the test suite code is organized: in particular, it depends on the use of the PO pattern. If the target test suite relies on the PO pattern, thread sleeps are located in the page object's methods. But these methods are usually called by more than one test method and so, to be sure that the refactored code did not introduce regressions or novel flakiness, we must run all the test methods that call the modified page object method (lines 3 and 4 of Algorithm 3). In a test suite without the PO pattern, instead, all the interactions with the web application are in the test methods, including thread sleeps. Therefore, to validate a replacement in this context, it is sufficient to run the modified test method (lines 7 and 8 of Algorithm 3).

3.3.2 Dealing with test dependencies during validation

As shown in the validation Algorithm 3, our approach manages both dependent and non-dependent test suites. A test method is called dependent when its execution result (pass or fail) depends on the order in which the test method is run in the test suite. If the test suite is always executed in the same, correct order, such dependencies may go unnoticed, but if we run a subset or a reordering of the test suite that breaks the dependencies, we will encounter failures. Thus, the presence of dependencies in the test suite to be refactored prevents from running a single test method for validation, since it requires other test methods to be executed before. To manage dependencies during the validation step, we employed one of the following two strategies, as appropriate:

1. **Using the warranted schedule**. A *warranted schedule* for a dependent test method *t* is a schedule that respects all the dependencies for *t*. Such schedules, if not known a priori, can be extracted from the dependency graph of the test suite, that can be computed, e.g., with tools like TEDD (Biagiola et al., 2019).
2. **Saving the application state**. if the characteristics of the system under test allow it, it is possible to save the application state required by each test method *t*, and restore it when it needs to run *t* for validation.

3.3.3 Effect of the thread sleeps position on the validation process

Another aspect that is important to consider in the validation step is the presence of thread sleeps in test fixtures. Test fixtures are utility methods named in the Selenium testing framework before methods and after methods. Before and after methods are mainly used to execute a portion of code before and after test methods. These are used to basically set up some variables or configuration to prepare the state of the application required for a test method and then to cleanup the state after the test execution ends. Lines 5 and 6 of Algorithm 3 manage this aspect: if the current thread sleep is located inside a test fixture, the validation set will contain the first test method of the containing class. Since usually in most testing frameworks (e.g., JUnit, TestNG) test fixtures can be executed before every method, after every method, before the whole class or after the whole class, by running a single test method from the containing class we are sure that also the test fixture is executed.

4 SleepReplacer tool

This section describes SleepReplacer, the tool that implements our approach. We implemented it as a Java application, built with the build automation tool Maven. It takes as input a Java test suite that uses the Selenium WebDriver framework to interact with web pages and TestNG[3] as unit testing framework. The tool, along with the three open-source test suites, is available at https://sepl.dibris.unige.it/SleepReplacer.php. We will now describe how we implemented each phase of the tool with a corresponding software component: the Model Builder, the Thread Sleep Replacer and the Validator.

4.1 Model Builder component

The Model Builder is the tool component that takes as input the original test suite and produces as output a model that contains information about thread sleep's locations, page accesses locations and their use in test methods. The model is represented using Java classes, and it is an instance of the meta-model expressed in Fig. 2. The Model Builder recovers the structure of the test suite classes (e.g., test classes and their methods, page objects and their methods) using reflection, and the location of thread sleeps and page accesses adopting static textual searches. If the test suite is not PO-based, all the information needed to build the model is already available. If, on the contrary, the test suite relies on the PO pattern, information about the usage of PO methods in test methods has to be collected: this is required because the Validator must know all the usages of an explicit wait in order to validate them. This is done by running an instrumented version of the test suite that generates a trace containing every test method and PO method execution in chronological order. From this trace, the Model Builder can reconstruct precisely which PO methods are used by the test methods.

To better clarify how the Model Builder component works, we provide a short step by step description of the underlying algorithm, both for PO-based and non-PO-based test suites.

[3] https://testng.org/doc/

Page object version:

1. running an instrumented version of the test suite that prints the names of the test methods, page object methods, and position of the thread sleeps. The output of this step is the *execution trace*;
2. by using reflection, building a model of the structure of the test suite's classes: for each class in the test suite (both page object classes and test classes) a corresponding class in the model is created, with its methods;
3. adding information about thread sleep locations in the model's classes;
4. for each thread sleep, searching in the code of the containing page object method the following page access, and add its location and type to the model;
5. relying on the execution trace built in step 1., creating the mapping of page object methods usages: we associate to each test method the list of page object methods it uses and vice versa.

No page object version:

1. creating a list of thread sleep positions inside test methods;
2. by using reflection, building a model of the structure of the test suite's classes;
3. adding information about the position of thread sleeps in the model's classes;
4. for each thread sleep, searching in the code of the containing page object method the following page access, and add its position and type to the model.

4.2 Thread Sleep Replacer component

The Thread Sleep Replacer is the core part of the tool, that performs the substitution of thread sleeps with explicit waits. To do its work, the Thread Sleep Replacer component relies on a set of replacement rules to decide which expected condition should be used depending on the type of page access is going to protect. As anticipated in the previous section, a replacement rule is a function $R : \{A_1, A_2, ..., A_n\} \rightarrow EC$ that maps a set of accesses A_n to an expected condition EC. There are two main categories of accesses to a web page: accesses that only read information from the page, and accesses that actively interact with the page. In our experience, accesses of the first type can be managed with a `visibilityOf` expected condition, accesses of the second type can be managed with a `elementToBeClickable` expected condition. Indeed, our years-long experience in E2E web testing (Ricca & Tonella, 2001; Leotta et al., 2016; Olianas et al., 2022; Leotta et al., 2015, 2016, 2021), and the results of our industrial previous work (Olianas et al., 2021) tell us that the great majority of web page interactions in a test suite can be managed using just two expected conditions: `visibilityOf` and `elementToBeClickable`. In our previous work, where 192 thread sleeps were replaced with explicit waits in a large, industrial test suite, the `elementToBeClickable` expected condition was used the 92% of the times. There are many other possible page accesses and corresponding expected conditions (see Sect. 2), but they are designed to manage specific cases (e.g., the presence of frames) which rarely happen. So for this work, we have applied the Pareto principle and limited ourselves to implement only a limited subset of rules, but with the awareness that these rules are able to eliminate a big portion of thread sleeps. The subset of rules we have implemented in SleepReplacer is the following:

1. R_1 : {getText, isEnabled, isDisplayed, getAttribute} → ExpectedConditions.visibilityOf
2. R_2 : {click, clear, sendKeys, selectByVisibleText, selectByIndex} → ExpectedConditions.elementToBeClickable
3. R_3 : {switchTo().alert()} → ExpectedConditions.alertIsPresent

We added the last rule (R_3) because, even if we did not meet them very frequently in our last refactoring work (Olianas et al., 2021), JavaScript alerts are quite common in Web applications to manage error notifications, that is an important aspect in web testing. Moreover, JavaScript alerts are not part of the DOM, so it is impossible to wait for them with any other expected conditions. It is also important to point out that SleepReplacer is parametric on the number and type of applicable rules, i.e., the set of rules used by SleepReplacer is extensible in case the test suite contains Web page accesses that can be managed with other expected conditions.

The Thread Sleep Replacer navigates the test suite model and, following the original order of execution of the test methods in the test suite, and the order in which thread sleeps are located inside single test methods, it applies the replacement rules to substitute each thread sleep with the appropriate explicit wait. After that, it saves the modified version of the file in the test suite project, compiles the project with Maven and calls the Validator component to check that the change in the code did not break the test method: if the validation fails, the Thread Sleep Replacer component will undo the last change in the code, restoring the initial thread sleep. We said that explicit waits, besides the expected condition, require also a maximum timeout to be waited for the expected condition to be verified. We used a default timeout of 10 s, based on our previous experiences, and never had problems caused by a too short timeout.

4.3 Validator component

Finally, the Validator is the component that runs the refactored code to check test methods that always fail and flakiness problems. The amount 'X' of validation executions is decided by the Software Tester. In case the Tester has no clues on the stability of the test suite, a possible solution is to base this choice on estimates: some authors reported that anecdotal evidence suggests to run tests 10 times (Palomba, 2019).

In Sect. 3, we said that the presence of dependencies in the test suite should be managed, and we presented two different options: running all the test methods required to satisfy the dependencies in the original order (first option), or saving the application's state required by each test method and restore it when a test method needs to be executed (second option).

Concerning the first option, our tool, when it comes to validate a dependent test *t*, will have to run its warranted schedule (i.e., a schedule that contains *t* and all the other tests required to satisfy *t*'s dependencies), instead of *t* alone, to avoid failures due to lack of previous test methods runs. If the dependencies between test methods are known it is sufficient to run the test methods in sequence. Otherwise, the right order of test methods to execute, can be calculated using TEDD (Biagiola et al., 2019). On the contrary, if the dependencies are not known and running a dependency detection tool like TEDD on the target test suite is impractical, a Tester can choose a conservative approach and, when a test method t has to be validated, he/she can run every test method that precedes t in the original order of the test suite.

Concerning the second option, our tool relies on Docker to create images of the state and running instances of the application under test. With respect to the previous option (i.e., warranted schedule execution), this solution is much more efficient from a performance perspective, but on the other hand is more complex to implement. In fact, it may not always be possible to save and replicate the state of the application under test: in some cases, the Software Tester may not have complete access to the application under test, but only to the test suite. In some other cases, the application may be distributed among different computational nodes, and saving and restoring its state may not be easy or possible. To save the state required for a test method t_n, we run all the tests t_1, t_2, t_{n-1} that precede it in the original order of the test suite, and we run them against an instance of the application under test executed in a Docker container. Then, we save a snapshot of the container state. When, during the validation step, we have to run t_n, we just need to launch a new Docker container with the state saved in the previous snapshot.

The first option is always applicable but less efficient. The latter is more efficient, since avoids to waste time in running test methods only to satisfy the order of dependencies, but it may not always be applicable, e.g., for applications whose state depends on many distributed components.

Finally, to validate thread sleeps used in test fixtures, the validator component runs the first test method in their containing class: in this way, we are sure that both the fixture and a test method that uses it have been validated.

4.4 Replacement example

To better explain our tool, let's present an example of how a thread sleep is detected, replaced and validated. We will use a snippet (Listing 7) from the test method AddUserTest from the Collabtive test suite, one of the test suites used in our experimental study (see next section), which does not employ the PO pattern.

```
driver.findElement(By.id("add_butn_member")).click();
Thread.sleep(1000);
driver.findElement(By.id("name")).clear();
```

Listing 7. Code snippet from the Collabtive test suite

The first line of code in Listing 7 clicks on the button "Add user" in Fig. 3, then waits 1000 ms for the form in Fig. 4 to be loaded. During step 1 of our approach, the Model Builder component stores in the model the line number of the thread sleep, along with the line number and type of the subsequent page access. The action performed is a *clear*, that clears the content of a text box, and so its page access type is WRITE.

Subsequently, the Thread Sleep Replacer component will consider the page access after the thread sleep, and will select an appropriate expected condition among the available ones. Since the action clears a text box, our tool will replace the thread sleep with an explicit wait that uses an elementToBeClickable expected condition (see Sect. 4.2). The

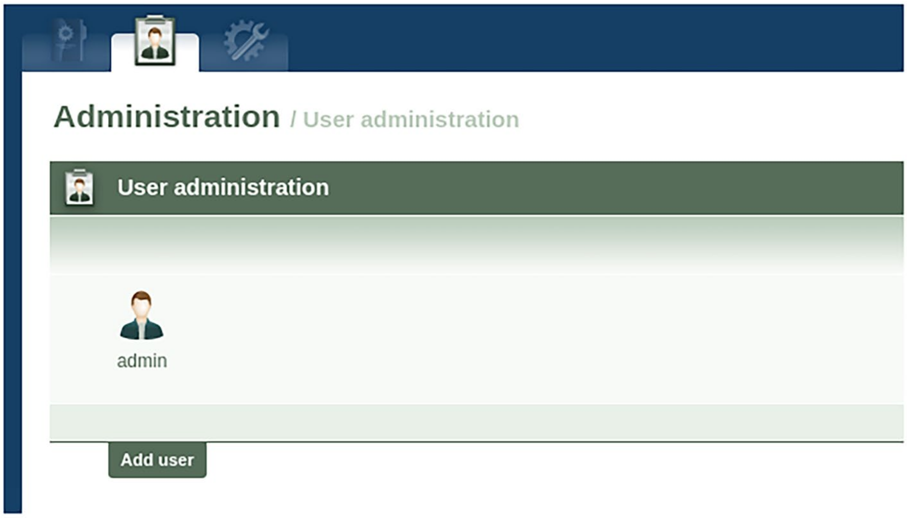


Fig. 3 Screenshot from the Collabtive web application

resulting code is given in Listing 8. (note that the *wait* object has already been initialized in the Before method associated with the test).

```
driver.findElement(By.id("add_butn_member")).click();
wait.until(ExpectedConditions.elementToBeClickable(By.id("name")));
driver.findElement(By.id("name")).clear();
```

Listing 8. Refactored code snippet from the Collabtive test suite

After the replacement, the Validator component compiles the modified test suite and tries to run the modified test method for a given amount of times. The Collabtive test suite has dependencies, and we already saved the state required for each test method to correctly run, so the Validator will launch the Docker container with the state required by AddUserTest before running it. After each correct execution, the container will be destroyed and recreated with the initial state (since the execution of the test method modifies it). If all the executions pass, the change is accepted and our tool moves to the next thread sleep. Otherwise, the change is reverted by restoring the thread sleep.

5 Empirical study

This section describes the design, experimental objects, research questions, metrics, validation framework, procedure, results and threats to validity of the empirical study conducted to evaluate SleepReplacer. We follow the guidelines by Wohlin et al. (2012) on designing and reporting empirical studies in software engineering. To allow the replication of the study, we published the tool along with the three open-source test suites at https://sepl.dibris.unige.it/SleepReplacer.php.

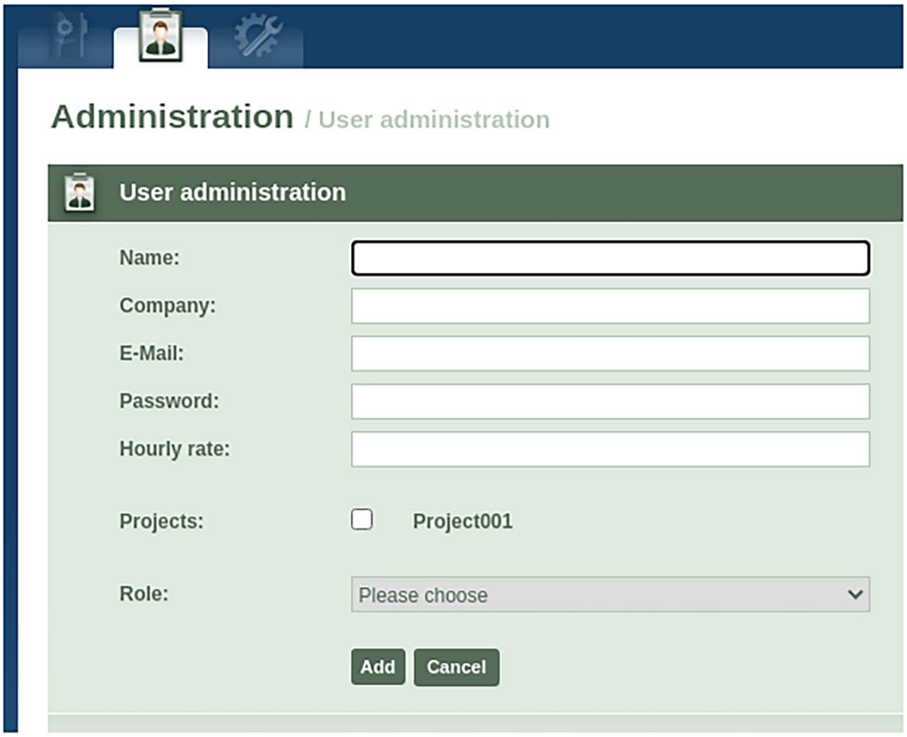


Fig. 4 Screenshot from the Collabtive web application

5.1 Study design

The *goal* of the empirical study is to *measure the overall effectiveness of* SleepReplacer *in replacing thread sleeps* with a particular focus on assessing: (1) the percentage of thread sleeps replaced by SleepReplacer, (2) the time required by SleepReplacer to complete such task, and (3) the human effort reduction deriving from its adoption.

The results of this study can be interpreted from multiple *perspectives*: Researchers, interested in empirical data about the effectiveness of a tool able to replace thread sleeps from existing Selenium WebDriver test suites; Software Testers and Project/Quality Assurance Managers, interested in evidence data about the benefits of adopting SleepReplacer in their companies. The *experimental objects*, used to experiment SleepReplacer are four test suites associated with four web applications described in the next section.

5.2 Experimental objects

To validate the proposed tool, several web applications and the corresponding test suites have been used. We used a large, medical web application (PRINTO) and three small open-source web applications (Addressbook, Collabtive, PPMA). Table 1 summarizes the main properties of the considered test suites. All the test suites are written in Java and use TestNG as testing framework and Selenium WebDriver to interact with the AUT.

Table 1 Properties of the experimental objects

Application	Test classes	test methods	Thread sleeps	Lines of code
PRINTO	17	169	146	10166
Collabtive	40	40	69	3108
Addressbook	27	27	10	2155
PPMA	23	23	82	2449

The PRINTO test suite has been developed in the context of a joint academia-industrial project by several junior Testers (Olianas et al., 2021). It has been also carefully refined so the source code quality is quite high. Moreover, it is executed automatically, every night, from several months without presenting significant problems. On the contrary, the other three test suites, have been developed/refined by PhD students in the context of academic research (in particular the preliminary versions have been developed in the context of an empirical study (Leotta et al., 2013) and then refined in further works such as (Olianas et al., 2021)). They are also of good quality but they have not been so refined over time, as in the case of the PRINTO test suite. Let us provide some additional details on the web applications under test and the corresponding test suites.

The Paediatric Rheumatology INternational Trials Organization (PRINTO)[4] is an international academic research network that co-ordinates international clinical trials in children with rheumatic and auto-inflammatory diseases. To collect information about patients from more than 500 centers worldwide, PRINTO has developed a large multi-page web application, written in PHP and JavaScript (approximately 100k PHP lines of code), mostly composed of forms that must be filled by the user (i.e., typically medical researchers). PRINTO test suite is composed by 169 test methods and 82 test fixtures, for a total of 251 test methods, and it is designed following the Page Object pattern. Moreover, some of the test methods are parametric: this means that they are executed multiple times with different input data during a single execution of the test suite. In total, 551 methods are executed for each run of the test suite. The original test suite we used in this experiment contains 146 thread sleeps.

Addressbook[5] is an open-source web application for contact management, that allows to store phone, address and birthday of user's contact list. It is written in PHP, uses MySQL as database and its test suite is composed by 27 test methods. The test suite contains in total 10 thread sleeps.

Collabtive[6] is an open-source web application for project management for small to medium sized businesses. It enables to manage the lifecycle of a project, which can be divided in tasks assigned to the different users. It is written in PHP and its test suite is composed by 40 test methods, that contain 69 thread sleeps.

PPMA (PHP Password Manager)[7] is an open-source web application, written in PHP, that allows to store passwords for different services. Its test suite is composed by 23 test methods, that contain 82 thread sleeps.

[4] PRINTO https://www.printo.it/

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

[6] Collabtive https://sourceforge.net/projects/collabtive/

[7] PPMA https://github.com/pklink/ppma

5.3 Research question, metrics, and procedure

Our study aims at answering the following four research questions:

RQ1: How many thread sleeps can SleepReplacer replace?
To answer our research question RQ1, it is necessary to count the original number of thread sleeps contained in each test suite associated with the selected web applications. Then, we have to count how many of them are replaced by SleepReplacer with explicit waits relying on the rules R_1, R_2, and R_3 described in Sect. 4. Finally, we compute the proportion of thread sleeps replaced by SleepReplacer out of the original total. Thus, the metric used to answer this question is the percentage of thread sleeps replaced.
RQ2: How long does it take SleepReplacer to replace the thread sleeps?
To answer our research question RQ2, it is necessary to measure the execution time required by SleepReplacer for replacing the thread sleeps. As a final measure, we provide the average time (expressed in minutes) for each test suite required by SleepReplacer to complete each individual thread sleep replacement.
RQ3: How much is the reduction of human effort using SleepReplacer?
To answer our research question RQ3, it would be necessary to have the time required for a human Tester to perform the replacement of the thread sleeps with and without SleepReplacer and computes the percentage of reduction. Unfortunately, not having available these data and not being able to design an experiment with experienced Testers specifically to answer this research question, we decided to provide an estimate-based answer. Basically, based on historical data we computed the average time a Tester takes to replace a single thread sleep. Since the execution of our tool takes place with negligible human effort (in background), we considered as human effort only that deriving from the thread sleeps that SleepReplacer was unable to replace. Subsequently, we computed the percentage of reduction comparing it to the total time calculated by multiplying the estimated time of a single replacement by the total number of thread sleeps contained in the original test suite.
RQ4: What is the effect of the SleepReplacer thread sleeps replacement on the overall test suite execution time?
To answer our research question RQ4, it is necessary to measure the execution time (in minutes) of each test suite before and after the thread sleeps replacement (i.e., before and after the execution of SleepReplacer). In this way, we can appreciate any benefits in terms of time reduction.

5.3.1 Settings for each web application

To run the experiment, we simply provided the four test suites as input to SleepReplacer and waited for the run to finish. The only parameter we had to decide is the number of validation runs for the step 2.(b) (Fig. 1).

In the PRINTO case, the large industrial test suite, it was sufficient just one validation run since we knew that the test suite was stable (as detailed in the answer to RQ3, we asked an independent Tester to replace all the thread sleeps for a previous version of PRINTO, and he rarely reported flakiness problems due to replacing the thread sleeps with explicit waits), thus we were able to run the tool and produce a valid test suite, without flakiness (note that a single run, when adopting the PO pattern, often implies multiple thread sleep validations as described below).

However, this was not the case for the other test suites since we did not have insights about the flakiness behavior of the test methods when the thread sleeps are replaced. In real industrial cases, this info is generally known (at least as an estimate) as human Testers have an idea of the flakiness behavior of their test suites. We tried to derive this information by manually eliminating about 10% of each application's thread sleep. The results are described for each web app in the following.

In the case of Collabtive, we needed 20 validation runs, since the replacement of some thread sleeps caused some test methods to fail non-deterministically, and this happened very rarely. For Addressbook we decided to do 20 validation runs, even if we did not expect flakiness problems, since its execution time and number of thread sleeps is very low. Indeed, we have been able to run the tool with 20 validation runs for each thread sleep in just 37 min. Finally for PPMA, since it had many more thread sleeps (82) and a longer execution time, we decided to set SleepReplacer with only 10 validation runs.

Two reasons that can explain the difference in terms of number of validation runs, between PRINTO and the open-source web applications, are the following: (a) the test suite quality is different, indeed as already said, the PRINTO test suite was carefully developed during a joint industrial project and is executed daily while the other three test suites were produced only for scientific purposes; (b) the PRINTO test suite adopts the PO design pattern while the other Web applications do not. Thus, in the case of the PRINTO test suite the thread sleeps are validated multiple times (even with a single test suite run), since the thread sleeps are inside the methods of the POs and there are multiple test methods calling them, while for other applications only once.

As a general guideline, given that the execution time is machine time (and not by far more costly human time), it would be advisable to repeat the validation as many times as possible, in order to minimize the probability of introducing flakiness.

For what concerns dependency management, PRINTO, the large, industrial test suite did not have dependencies, while the other three test suites did. We said that we have two options to run a dependent test method *t* during validation: (1) we can run all the test methods required to satisfy dependencies or (2) we save the state required by *t* to run correctly, and restore it when the tools need to run *t*. Since we had all the three applications under test installed in Docker containers and it is a more efficient solution, we opted for the second choice to manage dependencies in Collabtive, Addressbook and PPMA test suites.

Finally, we ran all the experiments on a laptop running Windows 10 with Intel Core i3 10110U CPU (maximum clock 4.10 GHz), 16 GB of RAM and SSD hard drive.

5.4 Results

RQ1: SleepReplacer Effectiveness in Replacing the Thread Sleeps

Table 2 shows: (1) the number of thread sleeps present in the various test suites for the four considered web applications (column Total), (2) the number of thread sleeps successfully replaced by SleepReplacer (column Replaced #), and finally (3) the percentage of the replaced thread sleeps with respect to the total number of thread sleeps (column Replaced %).

Table 2 Number of thread sleeps replaced by SleepReplacer on the four considered apps

Application	Total	Replaced	
		#	%
PRINTO	146	133	91%
Collabtive	69	56	81%
Addressbook	10	10	100%
PPMA	82	82	100%
Total	307	281	92%

In two cases out of four (i.e., for the Addressbook and PPMA web apps), the tool was able to successfully replace all the thread sleeps with the appropriate explicit wait. The minimum effectiveness of SleepReplacer was reached in the case of Collabtive where 56 out of 69 thread sleeps were replaced (corresponding to a still satisfying 81%). Finally, looking at the complex industrial PRINTO case study, SleepReplacer was able to manage 133 thread sleeps out of 146 (91%).

Let us now analyze why in two applications SleepReplacer was not able to replace all the thread sleeps. In the case of PRINTO, we observed 13 cases in which the tool was not able to complete the replacement. We analyzed the various cases and discovered that in most of the cases the problem was caused by dynamically loaded page elements via JavaScript: in these cases, the explicit waits inserted by SleepReplacer automatically are useless because they should have waited for another element rather than the one accessed directly by the test method. We give, in the following, a description of one of those cases, in our opinion, is the most interesting case. In PRINTO test suite, we have a test method that compiles a form with wrong values, tries to send it and checks if the web application responds with a specific error message, that is generated by a client-side script that checks the validity of the inserted data. SleepReplacer replaced the original thread sleep with an explicit wait waiting for the submission button to be clickable, and this change broke the test method because the obtained error message was different from what was expected. This happened because the thread sleep gave the client-side validation script enough time to complete, while the explicit wait, since the submission button is already clickable when the form is compiled, submitted the form before the validation script has finished. This resulted in a server-side error, that was different from the one expected by the test method, and thus the test method failed.

Similarly in the case of Collabtive, 13 thread sleeps remained after the execution of SleepReplacer. In this case, differently from PRINTO, most of the remained thread sleeps did not fail the test methods deterministically if replaced, but rather their replacement introduced some flakiness. A common situation is the one represented in Listing 9: we have a click on a web element, that causes the loading of another page, and we wait for it with a thread sleep. Then, we have some interactions that write some text in a form (the text “Task001”), and then another click on the form submission button.

```
driver.findElement(By.xpath("//div[3]/div/a[1]")).click();
Thread.sleep(1000);
driver.findElement(By.id("title")).clear();
driver.findElement(By.id("title")).sendKeys("Task001");
driver.findElement(By.xpath("//fieldset/div[6]/button[1]")).click();
```

Listing 9. Code snippet from the Collabtive test suite

Our tool, as it was built, replaced the thread sleep with an explicit wait that waits for the element located by the id "title" to be clickable, but during the validation the test occasionally failed on the last line (the click on the submission button). This happened because the form is loaded with an animation that makes it appear from top to bottom, and sometimes, when the "title" element is ready the animation is not ended yet, so the submission button is not clickable, and clicking it causes an `ElementNotInteractableException` to be thrown.

Thus, **to answer RQ1**, we can say that our tool SleepReplacer is effective in managing the automated migration — from thread sleeps to explicit waits — in the four considered test suites, since it was able to complete the replacement in the 92% of the cases, on average.

RQ2: Time Required for Replacing the Thread Sleeps

Table 3 shows the time required for replacing the thread sleeps using SleepReplacer. In the second column is reported for each web application the total time expressed in minutes required by a complete execution of SleepReplacer. Then in columns 3–4 and 5–6, we respectively analyze the time required for replacing each thread sleep considering respectively all the thread sleeps in the test suite and only the thread sleeps that SleepReplacer successfully replaced.

By looking at the table, it is evident that the thread sleeps contained in the three smaller test suites (i.e., the ones for the apps Collabtive, Addressbook and PPMA) required similar average times to be processed (i.e., in the order of 3 min each). Indeed, the execution time of SleepReplacer computed considering all the thread sleeps is in the range of 2.84–3.79 min for thread sleep; focusing only on the thread sleeps successfully replaced the time increases in the range 2.84-4.66 min for thread sleep. The lower range value is stable on the 2.84 value since corresponds to the case of PPMA where all the thread sleeps were successfully replaced. On the other hand, for Collabtive the value increases from 3.79 to 4.66 since about the 19% of the thread sleeps of the corresponding test suite are not replaced by SleepReplacer (note that Collabtive represents the worst case from this point of view, as described previously in Table 2).

On the contrary, in the case of the complex industrial PRINTO case study the time required to replace each thread sleep is higher: indeed it ranges from 9.56 min for thread sleep, when considering all the thread sleeps in the test suite, to 10.50 min for thread sleep when considering only the successfully replaced thread sleeps. This can be explained for three reasons: (1) the PRINTO test suite is based on the PO pattern and thus each thread sleep is contained in the PO methods; this leads to higher validation time since multiple test methods can use such PO methods and thus are executed; (2) the test methods are by far more complex than the ones of the other three web applications, so their execution time is by far higher; (3) unlike the PRINTO test suite, for the three open-source web applications (Addressbook, Collabtive, PPMA), we saved the application state required by each test method as explained in Sect. 5.3.1.

Thus, **to answer RQ2**, we can say that the time for successfully replace a thread sleep ranges in the interval 2.84–10.50 min with an average value of about 7 min. The actual values strongly depend on the complexity of the validation step (see Sect. 5.3.1), needed for assuring that the test suite provided in output by SleepReplacer do not present flakiness. This is true since the source code replacement executed by SleepReplacer (step 2.(a), Fig. 1) is clearly really fast. However, we can say that the obtained execution times are

Table 3 Time required (in minutes) for replacing the thread sleeps using SleepReplacer

Application	Total Time	Total		Replaced	
		# thread sleeps	Time for thread sleeps	# thread sleeps	Time for thread sleeps
PRINTO	1396	146	9.56	133	10.50
Collabtive	261.2	69	3.79	56	4.66
Addressbook	36.6	10	3.66	10	3.66
PPMA	232.6	82	2.84	82	2.84
Total	1926.4	307	6.27	281	6.86

absolutely acceptable, since the transformation performed by SleepReplacer has to be done only once, when the test suite is restructured.

RQ3: Percentage of reduction of human effort using SleepReplacer

Since we have not the manual thread sleeps replacement times, i.e., how long it would take an independent Software Tester to manually complete the thread sleeps replacement task for each web app, we decided to estimate such value using previous historical data. We have this information only for a previous version of the PRINTO test suite (the one with 196 thread sleeps). In that case, we asked to an independent Tester to substitute all the thread sleeps with explicit waits while recording both: (1) the time required for actually replacing the thread sleeps (i.e., the time he actually worked on the test suite code) and the validation time (i.e., the time spent to re-execute the test suite in order to check the absence of flakiness). To substitute the 196 thread sleeps, the Tester spent: (1) 556 min on the code (i.e., 2.84 min per thread sleep) and (2) 1309 min for the validation step (i.e., 6.68 min per thread sleep). In total, the overall time required to replace the thread sleep from that version of the PRINTO web app amount to 1865 min (i.e., 9.52 min per thread sleep).

In Table 4, we have used such computed values to estimate, and give an indication of, the human effort required to execute manually the thread sleeps replacement for the four considered applications (including PRINTO itself, since in this study we applied SleepReplacer to a different subsequent version).

In the case of the version of the PRINTO test suite used in this study, the total estimated is of about 23 h (1390 min); however since for a human Tester is by far more relevant to assess the actual time required while working on the test methods source code (since the validation process can be mainly done in background while working on other tasks; this is particularly true in case of longer and consecutive validation times), we can see that the time actually spent decreases to about 7 h (414 min). For the other applications the

Table 4 Estimated Time required (in minutes) by a human Tester for executing the thread sleep replacement task

Application	Total thread sleeps	Replacement Time	Validation Time	Total Time
PRINTO	146	414.64	975.28	1389.92
Collabtive	69	195.96	460.92	656.88
Addressbook	10	28.40	66.80	95.20
PPMA	82	232.88	547.76	780.64

values are proportional and still relevant: about 3 h for Collabtive and 4 h for PPMA, while Addressbook gets the shorter time of 28 min (but however, it is important to remember that it has only ten thread sleeps).

Thus, **to answer RQ3**, we can say that from the estimate performed with an independent Tester we found that replacing each thread sleep from the test code required, on average, about 3 min, while 10 min including also the validation time. In RQ1, we have said that SleepReplacer was able to replace overall 281 threads sleeps of 307 present in the four considered test suites. The replacement has been carried out fully automatically, without human intervention. Considering an average time required for a human of 3 min per thread sleep to replace (the most conservative estimate reported before), we have that: 307 * 3 = 931 min (where 307 is the total number of threads sleep in the four considered apps) represents the time required by a Tester to execute the complete thread sleep replacement task, fully manually, on the four test suites; 26 * 3 = 78 min is instead the time required by a Tester to replace only the thread sleeps that SleepReplacer was unable to replace. So even if this estimate is rough, we can conclude that the human effort reduction is very high (i.e., about 92%). Note that since in industrial test suites the number of thread sleeps could be in the order of hundreds or even thousands, the human effort savings due to the adoption of SleepReplacer in that contexts would be extremely relevant.

RQ4: Effect of the SleepReplacer thread sleeps replacement on the overall test suite execution time

Table 5 shows the execution time of the four considered test suites before and after replacing the thread sleeps using SleepReplacer. Columns 2 and 3 provide the total execution times (measured in minutes), respectively, for the original test suites with thread sleeps and for restructured one. Column 4 gives the percentage of reduction achieved thanks to the explicit wait adoption. From the table, it is evident that it is always advantageous to replace thread sleeps with the explicit waits: however, the magnitude of such positive effect is quite different considering the various web applications. The lower value has been observed in the case of Addressbook with a reduction of the 13%, while the complex industrial PRINTO case study benefited more, reaching a relevant 71% reduction. Note that having a 50% reduction (as in the case of PPMA) means halving the execution times.

The reason why the percentage reductions are so different lies probably in the fact that the number and frequency of the thread sleeps (i.e., number of thread sleeps per LOCs) in the considered test suites is not constant. Indeed, for instance Addressbook required only 10 thread sleeps to run properly, while others like PPMA, even if of comparable complexity, required by far more thread sleeps (in that specific case 82). Thus, assuming that the human Tester tuned optimally all thread sleep values, clearly having replaced more thread sleeps led to a more relevant reduction in the execution time. Indeed, explicit waits minimize automatically the time to wait, while when adopting thread sleeps, it is necessary to leave a small additional time margin that allows to manage any flakiness problems.

Table 5 Effect of SleepReplacer on the Test Suites execution time (in minutes)

Application	Time before replacement	Time after replacement	Reduction
PRINTO	126.23	37.11	71%
Collabtive	2.57	2.02	21%
Addressbook	0.72	0.63	13%
PPMA	2.13	1.02	52%

Thus, **to answer RQ4**, we can say that SleepReplacer is able to produce test suites that run always faster than their original counterparts. The benefits can vary a lot and depends on thread sleeps frequency; in our experiment from a 13% to a 71% reduction. More in detail, the magnitude of the percentage reduction, heavily depends on the initial impact of thread sleeps time on the total execution time: the % of thread sleeps time with respect to the total execution time of the test suite represents an upper bound for SleepReplacer. Thus, in the cases where the total sleep time is only a small fraction of the total test suite execution time, clearly, the benefits of using SleepReplacer are limited.

5.4.1 Discussion

In this subsection, we discuss the results obtained in our study, in order to highlight the benefits that the adoption of SleepReplacer can bring to the end-to-end testing process.

Results from **RQ1** show that SleepReplacer is able to replace automatically from 81 to 100% of the thread sleeps in a test suite. This is a strong point in favor of the adoption of SleepReplacer, since it tells us that the absolute majority of thread sleeps in a test suite can be replaced automatically. Moreover, we obtained such results using only three replacement rules; this has been done to maintain the approach as general as possible and to avoid ad hoc solutions tailored for the test suites used in the empirical evaluation. But in a real-world scenario, the Testers can easily add new replacement rules based on their specific knowledge of the test suite, in order to reach an even higher replacement rate. However, the results from **RQ4** show that even the test suite with the lowest replacement rate (Collabtive, 81%) obtained a significant time reduction (21%) from the use of SleepReplacer.

Results from **RQ2** highlight that the time to replace a thread sleep lies in the range of 2.84–10.50 min, with an average time of 7 min. The total times for replacing all the thread sleeps in a test suite range from 1396 min (approximately 23 h, for the PRINTO test suite) to 36.6 min. The high variability of total times depends on (1) the number of thread sleeps in the test suite, (2) the presence of the Page Object pattern, and (3) the number of validation runs required. If Testers want to employ SleepReplacer to improve a test suite, they must keep in mind these factors and try to estimate what the total time would be. However, even if the use of SleepReplacer may become infeasible on very large, test suites, with this study we showed that SleepReplacer can be used not only on small test suites, but also on real-world, medium-large sized test suites, as the PRINTO case.

Results from **RQ3**, although they are slightly hindered by the fact that the human time is only estimated, show that the adoption of SleepReplacer can lead to great time savings with respect to manual replacement of thread sleeps, when this task is faced. In fact, excluding validation time (that can be done in background while doing other tasks), every test suite in our study requires a human replacement time that goes from 28 to 414 min. Moreover, besides the reduction of the execution time of the test suite, another point in favor of the adoption of SleepReplacer is that manual work is error-prone, while our approach guarantees to produce a working test suite.

Finally, results from **RQ4** tell us that SleepReplacer achieves its goal, that is the reduction of the test suites execution time. In fact, comparing the execution time of the original versions of the test suites, with the time of the versions refactored by SleepReplacer, it is possible to observe a time reduction that goes from 13 to 71%. In our opinion, this is the strongest point in favor of the adoption of SleepReplacer, because even if the time-execution of SleepReplacer can be relevant, it is a “one-shoot’’ task, while the reduction of the execution time of the test suite can be appreciated every time the test suite is executed and can bring to substantial savings in developing environments where test suites are executed often.

5.5 Threats to validity

The main threats to validity affecting an empirical study are as follows: Internal, External, Construct, and Conclusion validity (Wohlin et al., 2012).

Internal Validity threats concern possible confounding factors that may affect a dependent variables: in this experiment the number of replaced thread sleeps (RQ1), the time required to replace them by SleepReplacer (RQ2), the time required by a human Tester for executing the thread sleep replacement task (RQ3), and the total test suite execution time (RQ4). Concerning RQ1 and RQ2, our tool is able to replace the thread sleeps with the explicit waits more used in practice. However, having test suite that requires different explicit waits would require to extend SleepReplacer to support them: note that SleepReplacer supports this kind of extension (basically it is sufficient to extend the rule list *R*), but this would impact on both RQ1 and RQ2 results. Concerning RQ3, as previously described, the whole calculation is based on an estimate and therefore is an approximation of the true value. We were forced to do an estimation because, in order to do a fair comparison, we could not replace the thread sleeps on our own, since we already well knew the test suites in study. In order to perform a real comparison with fair data, we would have needed experienced Testers but with no knowledge of our experimental objects. Concerning RQ4, as already describe in Sect. 5.3.1, the results are heavily related to the level of optimization adopted by the Tester during the definition of the thread sleeps times. In general, extending the wait times improve test suites stability but impact on the execution time. The values found in the four considered test suite are, in our opinion, reasonable, therefore the results obtained are generalizable to standard test suites.

External Validity threats are related to the generalization of results. All the four test suites for the web applications employed in the empirical evaluation of SleepReplacer are realistic examples covering a good fraction of the functionalities of the respective web apps. Moreover, the test suite for PRINTO has been developed in the context of an industrial project and includes 251 test methods: so its complexity is in line with standard test suites for web applications of average size.

Construct validity threats concern the relationship between theory and observation. Concerning RQ1, RQ2, and RQ4, they are due to how we measured the effectiveness of our approach with respect to the corresponding metrics. To minimize this threat, we decided to measure them objectively, in a totally automated way. Concerning RQ2 and RQ4 that can be influenced by the load of the computer executing respectively SleepReplacer and the test suite, to minimize any fluctuation, we averaged the obtained value three times. We have estimated that three times is sufficient since we noticed that the variance is minimal. Concerning RQ3, the threat is that the answer, being based on an estimate, could be prone to error. Another possible Construct validity threat is *Authors' Bias*. It concerns the involvement of the authors in manual activities conducted during the empirical study and the influence of the authors' expectations about the empirical study on such activities. To make our experimentation more realistic and to reduce as much as possible this threat, we adopted four test suites containing thread sleeps developed independently from other people, and existing before the development of SleepReplacer. Moreover, such test suites were not used during the development of SleepReplacer but only for its validation, in order to avoid any influence on the SleepReplacer implementation (e.g., including in SleepReplacer ad hoc solutions).

6 Related work

This work spans over several research directions, including the problem of flakiness in the context of software testing and techniques and approaches for test code refactoring.

6.1 Test flakiness

The problem of flakiness in regression testing has been faced by many authors in the last years. One of the most important and widely cited works has been published in 2014 by Luo et al. (2014). In this work, the authors classified the causes of flakiness by analyzing 52 open-source projects, and found that the top causes of flakiness are asynchronous waits (45%), concurrency (20%) and test order dependencies (12%). Their findings are confirmed by another study by Eck et al. (2019): in this work, the authors confirmed that asynchronous waits, concurrency and dependencies are the main causes of flakiness, but they found also other causes like too restrictive ranges in test assertions, platform dependency, test case timeout and test suite timeout.

Our experience confirms the results by Luo et al. and Eck et al. in particular on the high diffusion of the Async category as main reason of flakiness problems, which happens when a test method performs async calls without waiting the result. The solution to the async updates seems simple: sleeping the test for a bunch of milliseconds. This solution can make working the test method because it gives the app the time to update itself. However, how much the sleeping time should be it is totally unpredictable because it could depend on several factors, e.g., the network state, the total amount of available machine resources (i.e., CPU, RAM). As a consequence, every fixed delay can lead the test method to be more flaky and increasing its duration. The difficult is finding a balance between false negatives, i.e., when the test method fails because of a too low sleep and exaggerate sleep times.

There are many works in literature that identify thread sleeps as a source of instability, especially when they are ill-used, such as (Ahmad et al., 2019; Camara et al., 2021; Shukla, 2021). However, in another work, Presler-Marshall et al. (2019) analyzed different waiting strategies in Selenium E2E web tests, and concluded that thread sleeps gave the lowest flakiness, while explicit waits gave the highest. This tells us that, although in our experience explicit waits seem more reliable, they are not a silver bullet for resolving flakiness. Malm et al. (2020) recently published a short four-pages paper about automated analysis of flakiness mitigating delays. In particular, they present an automated approach to classify delays in test suites between sleeps of fixed duration (called thread sleeps in our work) and sleeps that use a polling/event-based approach (similar to the explicit waits presented in our work). Differently from our work their approach: (1) does not perform automated refactoring of the test suites to replace sleeps of fixed duration, but it performs only the analysis of the type of delays already present in the test suites, and (2) is not focused on Web testing so does not manage all the complexity required when interacting with a remote web system through a browser, as SleepReplacer does. From the analysis performed they concluded that sleeps of fixed duration are the most used. This result further motivates our work.

For what concerns flakiness in web testing, Moran et al. (2020) proposed a technique to locate the root cause of flakiness in test methods for web applications. This technique executes the test in different environmental configurations (e.g., network bandwidth, computational resources, screen size) in order to find the aspects that impact the most on test flakiness.

Some tool-based approaches to detect test flakiness has been proposed, such as DeFlaker by Bell et al. (2018), which uses code coverage to detect flaky tests: if a test changes its result, but it does not cover the modified code, then it is marked as flaky. Another tool-based approach is iFixFlakies by Shi et al. (2019). This approach is based on the idea that test suites composed of dependent test methods are flaky and therefore points to automatically fixing order-dependent test methods. The key insight in iFixFlakies is that test suites often already contain test fixtures methods that are executed before a test method (or a test class) to set up the initial state, and after a test method (or test class) to undo the test method actions with the goal of resetting the initial state of the application. Thus, iFixFlakies searches in a test suite test fixtures that make the order-dependent test methods pass and then use them for fixing the dependencies. Unlike these proposals, SleepReplacer does not take care of detecting or fixing flaky test methods, instead its goal is to eliminate thread sleeps without increasing flakiness.

6.2 Test code refactoring

According to Fowler's definition (Fowler, 2018), refactoring is "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure". Refactoring test code is a fundamental task to keep the test suite in line with production code, in particular when agile methods such as Extreme Programming (XP) are employed, as stated in Deursen et al., 2001). In this work, the authors list a set of code smells that may indicate poor quality of the tests, and a set of refactorings to eliminate such test smells. During the years, many proposals for automated refactoring tools have been made. Ikhsan and Candra (2018) proposed a tool-based approach to refactor web applications in order to improve accessibility: their tool automatically applies WCAG[8] guidelines for accessibility to a web application. Ying and Miller (2011) proposed a tool to automatically replace traditional forms with AJAX forms, to avoid refreshing the page when the form is submitted and hence improving the application's performance.

For what concerns test refactoring, in 2017 Stocco et al. (2017) proposed APOGEN, a tool-based approach able to automatically generate page objects for a test suite, in order to make it more maintainable. Their tool applies reverse engineering to the target application in order to extract a testing model, from which page objects are built. SleepReplacer has in common with APOGEN the usage of a testing model. Later on, Leotta et al. (2018) proposed PESTO, a tool-based approach that relies on aspect-oriented programming to automatically refactor a DOM-based test suite (i.e., a test suite that accesses page elements using DOM locators) to a visual-based test suite (i.e., a test suite that uses visual tools like Sikuli (Chang et al., 2010) to find elements in a web page). Finally, Cerioli et al. (2021) proposed TestWizard, a tool-based approach to asses test quality that aims to detect false-negative tests, i.e., test methods that pass when they should fail. This last work is not an automated refactoring tool, but rather a tool that employs automation to speed up the refactoring process.

[8] Web Content Accessibility Guidelines https://www.w3.org/WAI/standards-guidelines/wcag/

In common with these tools we have the fact that SleepReplacer can also be classified as a test code refactoring tool, since it modifies the performance of test methods but not their observable behavior (Fowler, 2018), even if its goal is completely different.

7 Conclusions and future work

In this paper, we have presented SleepReplacer, a tool-based approach able to automatically replace thread sleeps with explicit waits in E2E Selenium WebDriver test suites without introducing novel flakiness. The effectiveness of the proposed approach has been empirically evaluated using four test suites: one large industrial test suite and three smaller test suites built for open-source web applications. The empirical evaluation conducted to validate our approach showed that SleepReplacer is able to replace from 81 to 100% of thread sleeps in a test suite, resulting in a reduction of the execution time of the test suite that goes from 13 to 71%. The time required to replace a single thread sleep goes from 2.84 to 10.50 min, and since this is a completely automated process, the use of SleepReplacer can lead to great savings of human work.

As future work, we plan to add page inspection capabilities to SleepReplacer, enabling it to try some additional refactoring attempts. In fact, a limitation of SleepReplacer is that it relies only on the information available in the test suite code, that in some cases is insufficient to correctly replace a thread sleep. Furthermore, we would like to empower SleepReplacer with a mechanism that allows to reach a certain predefined level of stability in the final test suite produced by SleepReplacer. Indeed, as seen in the empirical study section, the higher the number of validation runs, the greater is the guarantee that the novel version of the test suite is free from flakiness. The idea would be to add an estimation mechanism to SleepReplacer able to estimate the time required for running the validation reaching a certain desired stability threshold (e.g., no flakiness in at least the 99% or 99.9% of the executions). Finally, another possible extension of SleepReplacer could be to add the functionality that allows us to calculate also the maximum timeout required by explicit waits.

Acknowledgements This work was carried out in the context of a collaboration between DIBRIS (University of Genova) and Printo (Gaslini Hospital). We want to show our gratitude to Dr. Nicolino Ruperto, Senior scientist of Printo for the support provided during the collaboration.

Funding Open access funding provided by Università degli Studi di Genova within the CRUI-CARE Agreement.

Declarations

Conflict of interest The authors declare no competing interests.

References

Ahmad, A., Leifler, O., & Sandahl, K. (2019). Empirical analysis of factors and their effect on test flakiness - practitioners' perceptions. arXiv:1906.00673

Ali, N., Engström, E., Taromriad, M., Mousavi, M., Minhas, N. M., Helgesson, D., Kunze, S., & Varshosaz, M. (2019). On the search for industry-relevant regression testing research. *Empirical Software Engineering, 24*. https://doi.org/10.1007/s10664-018-9670-1

Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., & Marinov, D. (2018). Deflaker: Automatically detecting flaky tests. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). pp. 433–444. https://doi.org/10.1145/3180155.3180164

Biagiola, M., Stocco, A., Mesbah, A., Ricca, F., & Tonella, P. (2019). Web test dependency detection. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, pp. 154–164. https://doi.org/10.1145/3338906.3338948

Camara, B., Silva, M., Endo, A., & Vergilio, S. (2021). On the use of test smells for prediction of flaky tests. In: Brazilian Symposium on Systematic and Automated Software Testing. pp. 46–54.

Cerioli, M., Lagorio, G., Leotta, M., & Ricca, F. (2021). Fight silent horror unit test methods by consulting a TestWizard. *Journal of Software: Evolution and Process,* e2396. https://doi.org/10.1002/smr.2396

Chang, T. H., Yeh, T., & Miller, R. C. (2010). GUI testing using computer vision. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 1535–1544.

Deursen, A., Moonen, L. M., Bergh, A., & Kok, G. (2001). *Refactoring test code*. NLD: Technical Report.

Eck, M., Palomba, F., Castelluccio, M., & Bacchelli, A. (2019). Understanding flaky tests: The developer's perspective. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA. pp. 830-840. https://doi.org/10.1145/3338906.3338945

Eda, R., & Do, H. (2019). An efficient regression testing approach for PHP web applications using test selection and reusable constraints. *Software Quality Journal*, *27*, 1383–1417. https://doi.org/10.1007/s11219-019-09449-2

Ekelund, E. D., & Engström, E. (2015). Efficient regression testing based on test history: an industrial evaluation. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 449–457. https://doi.org/10.1109/ICSM.2015.7332496

Fowler, M. (2018). *Refactoring: Improving the design of existing code*. Addison-Wesley Professional.

García, B., Gallego, M., Gortázar, F., & Munoz-Organero, M. (2020). A survey of the selenium ecosystem. *Electronics*, *9*(7). https://doi.org/10.3390/electronics9071067, https://www.mdpi.com/2079-9292/9/7/1067

Hossain, M., Do, H., & Eda, R. (2014). Regression testing for web applications using reusable constraint values. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops. pp. 312–321. https://doi.org/10.1109/ICSTW.2014.35

Ikhsan, I. N., & Candra, M. Z. C. (2018). Automatically: an automated refactoring method and tool for improving web accessibility. In: 2018 5th International Conference on Data and Software Engineering (ICoDSE). pp. 1–6. https://doi.org/10.1109/ICODSE.2018.8705894

Lam, W., Muşlu, K., Sajnani, H., & Thummalapenta, S. (2020). A study on the lifecycle of flaky tests. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. ICSE '20, Association for Computing Machinery, New York, NY, USA, pp. 1471–1482. https://doi.org/10.1145/3377811.3381749

Leotta, M., Biagiola, M., Ricca, F., Ceccato, M., & Tonella, P. (2020). A family of experiments to assess the impact of page object pattern in web test suite development. In: Proceedings of 13th IEEE International Conference on Software Testing, Verification and Validation (ICST 2020). IEEE, pp. 263–273. https://doi.org/10.1109/ICST46399.2020.00035

Leotta, M., Clerissi, D., Ricca, F., & Spadaro, C. (2013). Improving test suites maintainability with the page object pattern: an industrial case study. In: Proceedings of 6th International Conference on Software Testing, Verification and Validation Workshops (ICST 2013 Workshops). IEEE, pp. 108–113. https://doi.org/10.1109/ICSTW.2013.19

Leotta, M., Clerissi, D., Ricca, F., & Tonella, P. (2013). Capture-replay vs. programmable web testing: an empirical assessment during test case evolution. In: Proceedings of 20th Working Conference on Reverse Engineering (WCRE 2013). IEEE, pp. 272–281. https://doi.org/10.1109/WCRE.2013.6671302

Leotta, M., Clerissi, D., Ricca, F., & Tonella, P. (2016). Approaches and tools for automated end-to-end web testing. *Advances in Computers*, *101,* 193–237. https://doi.org/10.1016/bs.adcom.2015.11.007

Leotta, M., Ricca, F., & Tonella, P. (2021). Sidereal: Statistical adaptive generation of robust locators for web testing. *Software Testing, Verification and Reliability, 31*(3)*,* e1767. https://doi.org/10.1002/stvr.1767

Leotta, M., Stocco, A., Ricca, F., & Tonella, P. (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 2015). IEEE, pp. 1–10. https://doi.org/10.1109/ICST.2015.7102611

Leotta, M., Stocco, A., Ricca, F., & Tonella, P. (2016). ROBULA+: an algorithm for generating robust XPath locators for web testing. *Journal of Software: Evolution and Process (JSEP)*, *28*(3), 177–204. https://doi.org/10.1002/smr.1771

Leotta, M., Stocco, A., Ricca, F., & Tonella, P. (2018). PESTO: Automated migration of DOM-based Web tests towards the visual approach. *Journal of Software: Testing, Verification and Reliability (STVR), 28*(4), e1665. https://doi.org/10.1002/stvr.1665

Luo, Q., Hariri, F., Eloussi, L., & Marinov, D. (2014). An empirical analysis of flaky tests. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE'14, ACM, pp. 643–653. https://doi.org/10.1145/2635868.2635920

Malm, J., Causevic, A., Lisper, B., & Eldh, S. (2020). Automated analysis of flakiness-mitigating delays. In: Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test. AST '20, Association for Computing Machinery, New York, NY, USA, pp. 81–84. https://doi.org/10.1145/3387903.3389320

Moran, J., Augusto Alonso, C., Bertolino, A., de la Riva, C., & Tuya, J. (2020). FlakyLoc: Flakiness localization for reliable test suites in web applications. *Journal of Web Engineering*. https://doi.org/10.13052/jwe1540-9589.1927

Olianas, D., Leotta, M., & Ricca, F. (2022). MATTER: a tool for generating end-to-end IoT test scripts. *Software Quality Journal, 30*, 389–423. https://link.springer.com/article/10.1007/s11219-021-09565-y

Olianas, D., Leotta, M., Ricca, F., Biagiola, M., & Tonella, P. (2021). STILE: a tool for parallel execution of E2E webtest scripts. In: Proceedings of 14th IEEE International Conference on Software Testing, Verification and Validation (ICST 2021). IEEE, pp. 460–465. https://doi.org/10.1109/ICST49551.2021.00060

Olianas, D., Leotta, M., Ricca, F., & Villa, L. (2021). Reducing flakiness in End-to-End test suites: an experience report. In: A. C. R. Paiva, A. R. Cavalli, P. Ventura Martins, & R. Pérez-Castillo (Eds.), *Proceedings of 14th International Conference on the Quality of Information and Communications Technology (QUATIC 2021)* (vol. 1439, pp. 3–17). CCIS. Springer. https://doi.org/10.1007/978-3-030-85347-1_1

Page Object Model. https://www.selenium.dev/documentation/guidelines/page_object_models/. Accessed 10 January 2022.

Palomba, F. (2019). Flaky tests: Problems, solutions, and challenges. In: BENEVOL.

Presler-Marshall, K., Horton, E., Heckman, S., & Stolee, K. T. (2019). Wait wait. no, tell me: Analyzing selenium configuration effects on test flakiness. In: Proceedings of the 14th International Workshop on Automation of Software Test. AST '19, IEEE Press, pp. 7–13. https://doi.org/10.1109/AST.2019.000-1

Raghavendra, S. (2021). Waits. Apress, Berkeley, CA, pp. 129–142. https://doi.org/10.1007/978-1-4842-6249-8_10

Ricca, F., & Tonella, P. (2021). Analysis and testing of web applications. In: Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001, pp. 25–34. https://doi.org/10.1109/ICSE.2001.919078

Ricca, F., & Stocco, A. (2021). Web test automation: Insights from the grey literature. In: Proceedings of 47th International Conference on Current Trends in Theory and Practice of Computer Science. SOFSEM 2021, Springer.

SeleniumHQ. (2021). Web browser automation. https://www.selenium.dev/. Accessed 08 April 2021.

Shi, A., Lam, W., Oei, R., Xie, T., & Marinov, D. (2019). Ifixflakies: a framework for automatically fixing order-dependent flaky tests. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, pp. 545–555. https://doi.org/10.1145/3338906.3338925

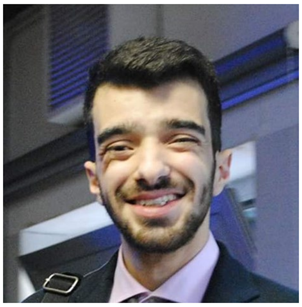
Shukla, S. (2021). *The protractor handbook*. Springer. https://doi.org/10.1007/978-1-4842-7289-3

Stocco, A., Leotta, M., Ricca, F., & Tonella, P. (2017). APOGEN: Automatic page object generator for web testing. *Software Quality Journal (SQJ)*, *25*(3), 1007–1039. https://doi.org/10.1007/s11219-016-9331-9

Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B., & Wessln, A. (2012). *Experimentation in Software Engineering*. Incorporated: Springer Publishing Company.

Ying, M., & Miller, J. (2011). Refactoring traditional forms into ajax-enabled forms. In: 2011 18th Working Conference on Reverse Engineering. pp. 367–371. https://doi.org/10.1109/WCRE.2011.51

Zhu, L., Bass, L., & Champlin-Scharff, G. (2016). Devops and its practices. *IEEE Software, 33*(3), 32–34. https://doi.org/10.1109/MS.2016.81



Dario Olianas is a PhD student at the University of Genova, Italy. He received his Master degree in Computer Science in 2019 with a thesis on automated testing of IoT systems. He is author or coauthor of some papers published in international journals and conferences/workshops.

Maurizio Leotta is a researcher at the University of Genova, Italy. He received his PhD degree in Computer Science from the same University, in 2015, with the thesis "Automated Web Testing: Analysis and Maintenance Effort Reduction". He is author or coauthor of more than 90 research papers published in international journals and conferences/workshops. His current research interests are in software engineering, with a particular focus on the following themes: Web, Mobile, and IoT application testing, functional test automation, empirical software engineering, business process modelling and model-driven software engineering.

Filippo Ricca is an associate professor at the University of Genova, Italy. He received his PhD degree in Computer Science from the same University, in 2003, with the thesis "Analysis, Testing and Restructuring of Web Applications". In 2011 he was awarded the ICSE 2001 MIP (Most Influential Paper) award, for his paper: "Analysis and Testing of Web Applications". He is author or coauthor of more than 100 research papers published in international journals and conferences/workshops. Filippo Ricca was Program Chair of CSMR/WCRE 2014, CSMR 2013, ICPC 2011, and WSE 2008. His current research interests include: Software modeling, Reverse engineering, Empirical studies in Software Engineering, Web applications and Software Testing.