Dipartimento di Informatica, Bioingegneria,
Robotica ed Ingegneria dei Sistemi

Test Quality Assurance for E2E Web Test Suites: Parallelization of Dependent Test Suites and Test Flakiness Prevention

by

Dario Olianas

Theses Series **DIBRIS-TH-2023-XX**

DIBRIS, Università di Genova

Via Opera Pia, 13 16145 Genova, Italy `http://www.dibris.unige.it/`

Università degli Studi di Genova

Dipartimento di Informatica, Bioingegneria,

Robotica ed Ingegneria dei Sistemi

Ph.D. Thesis in Computer Science and Systems Engineering
Computer Science Curriculum

Test Quality Assurance for E2E Web Test Suites: Parallelization of Dependent Test Suites and Test Flakiness Prevention

by

Dario Olianas

May, 2023

Dottorato di Ricerca in Informatica ed Ingegneria dei Sistemi
Indirizzo Informatica
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università degli Studi di Genova

DIBRIS, Univ. di Genova
Via Opera Pia, 13
I-16145 Genova, Italy
http://www.dibris.unige.it/

Ph.D. Thesis in Computer Science and Systems Engineering
Computer Science Curriculum
(S.S.D. INF/01)

Submitted by Dario Olianas
DIBRIS, Univ. di Genova
dario.olianas@dibris.unige.it

Date of submission: April 2023

Title: Test Quality Assurance for E2E Web Test Suites: Parallelization of Dependent Test Suites and Test Flakiness Prevention

Advisors:
Filippo Ricca
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università di Genova
filippo.ricca@unige.it

Maurizio Leotta
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università di Genova
maurizio.leotta@unige.it

Ext. Reviewers:
Angelo Susi
Fondazione Bruno Kessler, Trento
susi@fbk.eu

Mariano Ceccato
Università di Verona
mariano.ceccato@univr.it

Abstract

Web applications support a wide range of activities today, from e-commerce to health management, and ensuring their quality is a fundamental task. Nevertheless, testing these systems is hard because of their dynamic and asynchronous nature and their heterogeneity. Quality assurance of Web applications is usually performed through testing, performed at different levels of abstraction. At the End-to-end (E2E) level, test scripts interact with the application through the web browser, as a human user would do. This kind of testing is usually time consuming, and its execution time can be reduced by running the test suite in parallel. However, the presence of dependencies in the test suite can make test parallelization difficult.

Best practices prescribe that test scripts in a test suite should be independent (i.e. they should not assume that the system under test is already in an expected state), but this is not always done in practice: dependent tests are a serious problem that affects end-to-end web test suites. Moreover, test dependencies are a problem because they enforce an execution order for the test suite, preventing the use of techniques like test selection, test prioritization, and test parallelization.

Another issue that affects E2E Web test suites is test flakiness: a test script is called flaky when it may non-deterministically pass or fail on the same version of the Application Under Test. Test flakiness is usually caused by multiple factors, that can be very hard to determine: most common causes of flakiness are improper waiting for async operations, not respected test order dependencies and concurrency problems (e.g. race conditions, deadlocks, atomicity violations). Test flakiness is a problem that affects E2E test execution in general, but it can have a greater impact in presence of dependencies, since 1) if a test script fails due to flakiness, other test scripts that depend on it will probably fail as well, 2) most dependency-detection approaches and tools rely on multiple executions of test schedules in different orders to detect dependencies. In order to do that, execution results must be deterministic: if test scripts can pass or fail non-deterministically, those dependency detection tools can not work.

This thesis proposes to improve the quality assurance for E2E Web test suites in two

different directions:

1. *enabling the parallel execution of dependent E2E Web test suites in a optimized, efficient way*
2. *preventing flakiness by automated refactoring of E2E Web test suites, in order to adopt the proper waiting strategies for page elements*

For the first research direction we propose STILE (teST suIte paralLElizer), a tool-based approach that allows parallel execution of E2E Web test suites. Our approach generates a set of test schedules that respect two important constraints: 1) every schedule respects existing test dependencies, 2) all test scripts in the test suite are executed at least once, considering all the generated schedules.

For the second research direction we propose SleepReplacer, a tool-based approach to automatically refactor E2E Web test suites in order to prevent flakiness.

Both of the tool-based approaches has been fully implemented in two functioning and publicly available tools, and empirically validated on different test suites.

Table of Contents

Chapter 1 Introduction **6**

Chapter 2 Background **10**

2.1 Web applications 10

2.1.1 Overview of Web Application's Frontend 11

2.1.2 CSS 12

2.1.3 JavaScript 12

2.1.4 AJAX 13

2.1.5 Single Page Applications 13

2.2 Introduction to Software Testing 14

2.2.1 The Pyramid of Testing 14

2.3 End-to-End Web Testing 16

2.3.1 Page Object Pattern 18

2.3.2 Waiting Strategies for E2E Web Test Scripts 19

2.4 Test Dependencies 23

2.5 Test Flakiness 24

2.6 Parallel Execution of E2E Web Test Suites 25

2.6.1 Selenium Grid 26

Chapter 3 Related Work **28**

3.1 Test Dependencies 28

3.2 Test Flakiness . . . 33
3.3 Test Parallelization . . . 35

Chapter 4 STILE: Optimized Parallelization of Dependent E2E Web Test Suites **37**
4.1 Challenges of Parallelizing Dependent Test Suites . . . 37
4.1.1 Test Dependency Graph and Warranted Schedules . . . 38
4.1.2 Managing the Web Application State . . . 39
4.2 Baseline Approach for Parallel Execution of Dependent Test Suites with Selenium Grid . . . 40
4.3 Our Proposal . . . 41
4.3.1 The Main Phases of STILE . . . 43
4.3.2 Implementation of STILE and Deployment . . . 46
4.4 Empirical Evaluation . . . 48
4.4.1 Study Design . . . 49
4.4.2 Research Questions, Metrics, and Experimental Procedure . . . 50
4.4.3 Results . . . 51
4.4.4 Summary of the Findings . . . 62
4.4.5 Threats to Validity . . . 63

Chapter 5 Test Flakiness Prevention: SLEEPREPLACER **65**
5.1 Motivation . . . 65
5.2 The Role of Waiting Strategies in E2E Test Flakiness . . . 66
5.3 Our Proposed Approach . . . 67
5.3.1 Step 1 - Model Building Phase . . . 68
5.3.2 Step 2.(a) - Thread Sleep Replacement Phase . . . 70
5.3.3 Step 2.(b) - Validation Phase . . . 72
5.4 SLEEPREPLACER: Approach Implementation . . . 74
5.4.1 Model Builder Component . . . 74
5.4.2 Thread Sleep Replacer Component . . . 76

5.4.3 Validator Component . . . 77
5.4.4 Replacement Example . . . 78
5.5 Empirical Study . . . 80
5.5.1 Study Design . . . 80
5.5.2 Experimental Objects . . . 80
5.5.3 Research Question, Metrics, and Procedure . . . 82
5.5.4 Results . . . 84
5.5.5 Threats to Validity . . . 90

Chapter 6 Conclusions **92**
6.1 Future work . . . 93

Bibliography **95**

Chapter 1

Introduction

In today's world, Web applications are widely used and their capabilities keep growing with time. Differently from old, static Web pages, modern Web applications provide a high level of user interaction, and are becoming the preferred way to develop an application, since developing a Web application is way easier than developing and supporting multiple versions of an application for different platforms [SA20].

Given their diffusion, assuring quality of Web applications becomes a task of paramount importance, as ever more business-critical services rely on them. Many testing approaches and techniques have been proposed and developed to improve the reliability of such systems. Among them, End-to-end (E2E) testing is an approach able to test the application as a whole to ensure that the system behaves as expected. It consists in automating the set of manual operations that a human user would perform on the pages of the Web application, interacting with the pages through a Web browser [LCRT16b]. The interaction with the browser is performed through different testing frameworks: one of the most used, and the one that we will use in this thesis, is Selenium WebDriver. Since E2E test scripts perform the same actions that a user would do, E2E testing provides a high fault detection capability, but on the other hand E2E test scripts tend to be more fragile with respect to other kinds of testing like unit and integration testing [RLS19]. In fact, even if writing E2E Web test scripts is not a very difficult task for the developer thanks to testing frameworks, the execution of a E2E test script is a very complex task that involves at least a browser, the Web application front-end and back-end, the framework that manages the interaction with the browser and the test script program itself. Moreover E2E testing, depending on the size of the application under test, can become very time consuming [GD13], and therefore developers may want to speed up the execution by the use of test parallelization, a testing technique where a test suite is divided in parts that are run in parallel, at the same moment. But given the aforementioned complexity of the execution of E2E test scripts, running a E2E test suite in parallel is a complex task. In particular, test parallelization is prevented by the presence of test dependencies in the test suite [ZJW+14b, RS21]. The problem of test dependencies is

not limited to the E2E Web testing field, but it also affects other kinds of testing. Best practices prescribe that test scripts in a test suite should be independent, but this is not always true in practice, since testers might rely on the application state set by previous tests when designing a test script [RS21, GSHM15, ZJW+14a, BKMD15, GBZ18]. Test dependencies are particularly problematic when they are not known to the developers, because if dependencies are broken without knowing it, they may cause unpredictable failures that are hard to troubleshoot [LHEM14]. For this reason, different approaches have been proposed in literature to detect unknown dependencies in test suites [GBZ18, BSM+19, ZJW+14a]. We will describe them more in detail in subsequent chapters of this thesis, but the important thing to know is that these approaches require to run test scripts multiple times in different orders and check their results in order to detect dependencies. Therefore, the application of these approaches is prevented by the presence of test flakiness. A test script is *flaky* when it may non-deterministically pass or fail on the same version of the Application Under Test (AUT), i.e. leading to different results in different runs on the same AUT without any change in both the app and test code [EPCB19, ZPSH20]. The flakiness problem is very insidious for companies because: (1) it makes lose confidence in the results of the execution of the test suites with false alarms, (2) it increases deployment/release times, and generally, (3) it increases development costs [ZPSH20].

The main goal of this thesis is to improve the efficiency of the execution of E2E Web test suites from two different perspectives: reducing the execution time of the test suite and assuring the correctness of the results. Although they are related to the same main goal, these two aspects of E2E Web test efficiency require to be addressed with different approaches and ideas. To reduce the execution time of a test suite, not only in the Web field, a commonly used strategy is test parallelization, i.e. running more test scripts in parallel rather than sequentially. Differently from other testing fields like unit testing, parallelization of E2E Web test suites poses different challenges to the tester, like the management of the state of the application under test and the interaction of different tests with different browsers. To ensure the correctness of the execution results, instead, the tester should ensure that the test scripts are not flaky, i.e. that they always pass when the application under test behaves correctly and they always fail when the application under test does not behave correctly, without non-deterministic behaviours. The absence of flakiness can be seen as a prerequisite for test parallelization, and for test execution in general since, as we said before, the presence of flakiness makes the developers lose confidence in test results and generally worsens the testing process. Moreover, as we briefly introduced before, and as we will explain better in Section 5.1 flakiness can prevent the execution of many dependency detection tools, that are required to enable the parallelization of dependent Web test suites. Therefore, we divide the main goal of this thesis in two sub-goals, that will be addressed by two different tool-based approaches. The two sub-goals of this thesis are:

1. Enabling parallelization of E2E Web test suites in presence of dependencies
2. Preventing flakiness by using the proper waiting strategies for page elements loading

To achieve the first goal we propose STILE (teST suIte paralLElizer), a tool-based approach that allows parallel execution of E2E web test suites. STILE generates a set of test schedules that respect two important constraints: 1) every schedule respects existing test dependencies, and 2) all test scripts in the test suite are executed at least once. Moreover, STILE optimizes the execution by running only once the test scripts that are shared among the schedules, relying on Docker to easily replicate the required state of the different instances system under test. We implemented our approach in a tool and empirically evaluated it on eight E2E Web test suites. We compared STILE with both the sequential execution of the test suite and a parallel, but non-optimized execution using Selenium Grid, and our results show that STILE can reduce the execution time up to 75% w.r.t. the sequential execution and up to 54% w.r.t. Selenium Grid. Moreover, STILE provides a reduction in the energy consumption up to 75%.

To achieve the second sub-goal we propose SLEEPREPLACER, a tool based approach to automatically refactor E2E Web test suites in order to prevent flakiness. To present SLEEPREPLACER we need to briefly talk about waiting strategies in E2E Web test suites. Asynchronous calls in E2E test suites are usually managed in two ways: thread sleeps and explicit waits. The first pause the execution of the test for a given amount of time, giving the page the time to load. Explicit waits, instead, actively poll the browser to know when a specific element is loaded. Usually, best practices prescribe not to use thread sleeps, because they are inefficient (if the waiting time is too long) and can be a source of flakiness (if the waiting time is too short), but they are still widely used in practice. Refactoring a test suite from thread sleeps to explicit waits can be a difficult and time consuming task, since each change should be validated to be sure it didn't introduce flakiness. In Section 2.3.2 we provide a more detailed description of waiting strategies for E2E Web test scripts.

Our tool SLEEPREPLACER is able to automatically replace thread sleeps with explicit waits without introducing novel flakiness, following a step-by-step approach where every thread sleep is replaced and validated in isolation, relying on multiple executions of the modified test script to be sure that the execution is not flaky. We empirically validated SLEEPREPLACER on four different test suites, and we found that it can correctly replace in 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%).

Both of our tool-based approaches have already been published in scientific literature. In detail, in 2021 we published a conference paper [OLRV21], presented at the *14th International Conference on the Quality of Information and Communications Technology (QUATIC 2021)*, that describes a manual approach to replace thread sleeps with explicit waits in E2E test suites. Later on, we improved and automated this approach, implementing it in the SLEEPREPLACER tool. The paper [OLR22] that describes SLEEPREPLACER has been published in 2022 on the *Software Quality Journal (SQJ)*.
For what concerns STILE, in 2021 we published a conference paper [OLR$^+$21b], presented at the *IEEE International Conference on Software Testing, Verification and Validation 2021 (ICST*

2021). In this paper, we describe the key concepts and ideas of our approach and provide a theoretical estimation of how much it will reduce the execution time fo the test suites. A journal paper that describes more in detail the approach and, most importantly, provides an empirical evaluation of the approach performed using actual execution of our tool, has been submitted to the journal *ACM Transactions on Software Engineering and Methodology (TOSEM)* and is undergoing revision.

This thesis is organized as follows:

- in **Chapter 2** we give an introduction on Web applications and E2E Web testing, and we present the key concepts used in this thesis, namely test dependencies, test flakiness and test parallelization;
- in **Chapter 3** we provide an overview of existing literature related to the topics of this thesis;
- in **Chapter 4** we present STILE, a tool-based approach to address the first research direction of this thesis: the optimized parallel execution of dependent E2E Web test suites. After presenting the key ideas behind STILE, along with the baseline approach with which we compared it, we will describe our approach and its implementation. After that, in Section 4.4 we will provide the empirical evaluation of our approach using 8 E2E Web test suites;
- in **Chapter 5** we present SLEEPREPLACER, a tool-based approach to address the second research direction of this thesis: flakiness prevention in E2E Web test suites. After a motivation for the approach, that explains why it is related to dependency-aware parallelization, we will describe the steps that compose our approach and their implementation. After that, in Section 5.5 we will provide the empirical evaluation of our approach using 3 small, open source test suites and one industrial, large test suite;
- finally, **Chapter 6** concludes this thesis with a summary of its contributions and some possible directions for future work.

Chapter 2

Background

In this chapter, we present the key concepts that are used in this thesis. After an introduction about web applications and related technologies, we will give a brief overview on software testing. Then, we will introduce the topics of software testing that we used in this thesis, namely End-to-end (E2E) testing, test dependencies, test flakiness, and test parallelization for E2E Web test suites.

2.1 Web applications

A web application is a type of software application that is accessed through a web browser and runs on a web server. It typically adopts a client-server architecture, where the client (the web browser) interacts with the server to request and receive data. A classical web application is composed of three parts:

- The **frontend**, also called the client-side, is the interface that users interact with. It is usually built using HTML (HyperText Markup Language), CSS (Cascading Style Sheets), and JavaScript. The frontend is responsible for handling user interactions, and making requests to the backend for data.
- The **backend**, also called the server-side, is the part of the web application that runs on the server and processes requests from the frontend. It is responsible for handling business logic, processing data, and interacting with databases or other external services. The backend is typically built using programming languages such as Python, Ruby, PHP, or Node.js.
- The **database** is where the web application stores and retrieves data. It can be a relational database, such as MySQL or PostgreSQL, or a NoSQL database, such as MongoDB or

Cassandra. The backend interacts with the database to store and retrieve data, such as user information, product data, or other application data.

Since this work focuses on E2E testing, we will now introduce some basic concepts and technologies used in Web application's frontend.

2.1.1 Overview of Web Application's Frontend

2.1.1.1 HTML and the Document Object Model (DOM)

HTML (HyperText Markup Language) is the standard markup language used create Web pages, both for static websites and dynamic web applications. An HTML document contains the text of the actual page that will be displayed in the browser enclosed in tags, that represent the different kind of elements that the page will contain (e.g., `<img>` for images, `<p>` for paragraphs, `<table>` for tables). In a standard Web page, HTML tags are nested: there is a main `<html>` tag that contains the whole page, and a `<body>` tag that contains the main content of the page, that is visible to users.

Nested tags in HTML documents define a hierarchical tree structure called Document Object Model (DOM). The DOM represents each element of a web page as a node in a tree structure, where the elements contained inside another elements are represented as children of the containing element. Web browsers offer an API (Application Programming Interface) to access, manipulate and interact with elements in a web page. This is fundamental both for modern web applications, whose pages are frequently dynamically changed via JavaScript (see Section 2.1.4), and for E2E web testing, since E2E tests rely on the DOM API to interact with the pages of the Web Application Under Test (WAUT). Figure 2.1 shows an example of simple HTML document, and Figure 2.2 shows the corresponding DOM tree structure.

```
<html>
<head>
<title>Example</title>
</head>
<body>
<div>
    This is an example page. For more information, visit
    <a href="https://en.wikipedia.org/wiki/Document_Object_Model">Wikipedia</a>
</div>
</body>
</html>
```

Figure 2.1: Example of HTML document

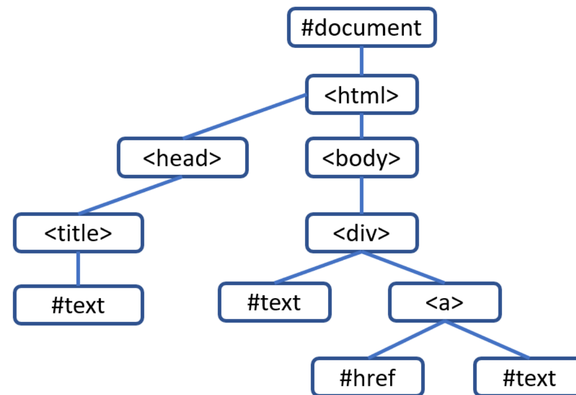

Figure 2.2: DOM tree of the example HTML document

2.1.2 CSS

CSS (Cascading Style Sheets) is a style sheet language used to define the appearance of a HTML document. CSS allows to define how elements of the page should be displayed, i.e. layout, fonts, colors and other visual properties. CSS uses selectors to target HTML elements, and uses rules to define how the target element should be displayed. For example, the statement

```
p {
text-align: center;
font-color: red;
}
```

says that paragraph elements (`<p>` tag in HTML) must have the text aligned to the center and the font color must be red.

To separate content from presentation, usually the CSS for a web page is contained in a separated .css file

2.1.3 JavaScript

JavaScript is a scripting language that is primarily used to make interactive web pages. JavaScript adds interactivity to classic HTML pages by enabling complex animations, pop-up elements and dynamic changes in the page. JavaScript is executed in the client browser, and relies on the DOM API offered by the browser to interact with page elements and intercept events, such as clicks made by the user on page elements. JavaScript can listen to such events using event listeners and execute the proper code (event handler or callback) when a certain event is detected.

In the context of web application frontends, JavaScript is mainly used for:

1. Adding interactivity to web pages, for example allowing client-side form validation (the data entered in a form by the user can be validated before sending it to the server), animations, event handling
2. Manipulating the DOM in order to modify the structure of a page in real-time, with no need to reload the page
3. Making HTTP requests to remote servers, in order to fetch new content without triggering a reload of the page. This last feature is the core of AJAX, a fundamental technology for modern web applications that we will present in the following of this section.

2.1.4 AJAX

AJAX [G+05], short for Asynchronous JavaScript and XML, is a web development technique that enables asynchronous communication between a web browser and a web server. AJAX allows to dynamically load new content on the page without reloading the whole page: the new content is requested from the server through AJAX and displayed by modifying the DOM of the page through JavaScript. This results in a faster and smoother experience for the user, that does not have to wait for the whole page to be reloaded. Today, a huge number of web applications relies on AJAX, for example social networks, web-based e-mail clients, online productivity suites and much more.

The advent of AJAX has drastically changed the web development world, enabling a new paradigm for developing Single Page Applications (SPAs), that we will describe in the next subsection.

2.1.5 Single Page Applications

A Single Page Application (SPA) is a type of web application that loads a single HTML page, and updates its content dynamically through AJAX. In contrast to traditional multi-page web applications, where each user action triggers a reloading of the page, SPAs load a single page initially and then dynamically update its content according to the user actions. Moreover, to enable browsing the application while keeping a meaningful URL, SPAs rely on client-side routing, which is a technique where the routing logic is managed on the client-side rather than by the server. When using client-side routing, JavaScript intercepts the request, retrieves the required data from the server and displays it on the page, without triggering a full page reload.

In this section, we showed that web applications are highly modular pieces of software, whose behavior does not depend on a single component, but rather on the interaction of different components. Therefore, in order to assess the quality of a web application, testing its components in isolation is not sufficient, but instead its required to test also the entire application as a whole. This is the goal of End-to-End web testing, that we will describe in Section 2.3

2.2 Introduction to Software Testing

According to the IEEE/ANSI 1059 definition [IEE94], software testing is the process of analyzing a software item to detect the differences between existing and required conditions (that is defects/errors/bugs) and to evaluate the features of the software item. Testing is performed by running test cases against the system under test (SUT) and collecting their results. According to the definition of the IEEE/ISO/IEC Systems and software engineering vocabulary [IEE10], a test case is a set of inputs, execution conditions and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. The execution of a test case returns a result, that can be PASS (the system behaves as required) or FAIL (the system does not behave as required). The set of test cases that validate a system under test is called a *test suite*.

Software testing can be performed at different levels, usually represented using the pyramid of testing, that we introduce in the next subsection.

2.2.1 The Pyramid of Testing

First introduced by Cohn [Coh10], the pyramid of testing, represented in Figure 2.3 is a useful concept to understand the different levels at which software testing can be performed.

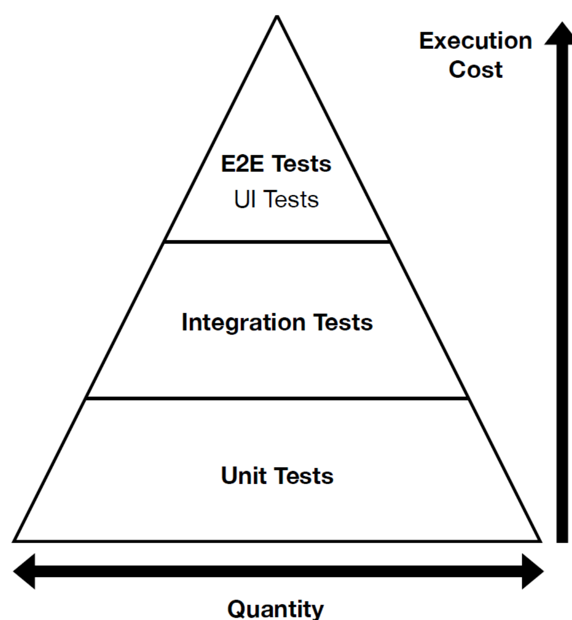

Figure 2.3: The pyramid of testing (picture from [Coh10])

The pyramid of software testing consists of three levels, with each level representing a different type of test, as follows:

- **Unit tests** are the foundation of the pyramid and form its base. These tests focus on testing individual units of code, such as functions or methods, in isolation to verify that they work correctly. Unit tests are typically written and executed by developers during the development process to catch bugs and ensure that individual units of code are functioning as expected. They are usually automated, fast to execute, and provide quick feedback on the correctness of the code at a granular level.
- **Integration tests** sit in the middle of the pyramid. These tests focus on testing the integration and interaction between different units or components of the software. Integration tests verify that different units or components of the software work correctly when combined together and interact with each other as intended. Integration tests may involve testing the interaction between modules, APIs, services, or databases, and can be automated or manual depending on the complexity of the integration points.
- **End-to-End (E2E) tests** are at the top of the pyramid and form the smallest portion. These tests focus on testing the entire software system as a whole, including the interactions between all the components of the software, from the user interface to the backend systems. E2E tests simulate real-world scenarios and user interactions to verify that the software system behaves correctly from end to end. E2E tests are typically slower and more complex to set up and maintain compared to unit and integration tests, and they may involve tools and frameworks that can simulate user interactions and test across multiple components.

The pyramid of software testing promotes a testing strategy where the majority of tests are focused on the lower levels of the pyramid, i.e., unit and integration tests, as they are typically faster, cheaper, and provide quicker feedback during the development process. E2E tests, although important for testing the system as a whole, are typically fewer in number and more time-consuming and expensive to execute. A well-balanced and effective testing strategy that follows the pyramid of software testing helps catch bugs early in the development cycle, improves software quality, and reduces the overall cost and effort of testing.

But as we said in the previous section, E2E testing is particularly important in the web testing field, because is the one that can verify the actual behaviour of the application, the behaviour that the user will see. Moreover, since E2E testing is the most time consuming level of testing, it is also the one that benefits the most from testing techniques that enable reducing the execution time of the test suite, such as test parallelization. Therefore, this is the testing level on which our work focuses on. In the next section, we will provide an overview on how E2E testing is performed in the web testing field.

2.3 End-to-End Web Testing

End-to-End (E2E) testing is a black-box testing technique that aims to test a system as a whole, as a human tester would. An E2E test case is composed by a sequence of actions performed on the web application GUI in order to exercise the web application functionalities, and some assertions used to assess if the application behaves as expected. A test suite is a collection of test cases which is used to test the various functionalities of the web application under test (WAUT). Different approaches and tools exist to implement and execute E2E Web test cases, such as Capture-Replay Testing, Programmable Testing [LCRT16a], and more recently Natural Language Testing [LRSM22]. In this thesis, we will focus on Programmable Testing. Programmable web testing consists in writing test scripts in a general purpose programming language, like Java or Python, relying on testing frameworks like Selenium WebDriver [GGGMO20]. These frameworks offer functionalities to easily interact with a web application, e.g. functions to click an element, enter text in an input field, read text from an element, wait for an element to be loaded and much more.

Listing 2.1 shows a test script from an E2E test suite we used as experimental object. Such test script exercises the page that is shown in Figure 2.4. After instantiating the driver (Line 1), the test opens the homepage of the application under test (Line 2), clears the fields login and password (in case some text is already present, respectively Lines 3 and 5), inserts the respective values (Lines 4 and 6) and clicks the login button (Line 7). Finally, the assertion at Line 8 checks if the login was successfully by comparing the displayed username (the one returned by the .getText() invocation) with the expected one (”Firstname001 Name001”). Web elements in the web page are found using *locators*: strings used in E2E test scripts to identify the position of elements of the web page [LSRT15] in the GUI. Common examples of locators are IDs, CSS selectors and XPaths. In our example, locators are the first argument of the `findElement` method calls. For example, the expression “By.id(”login”)” indicates that the test script is looking for a web element in the page whose attribute “id” is equal to “login”.

```java
1 WebDriver driver = getDriver();
2 driver.get("http://localhost/claroline11110/claroline/index.php");
3 driver.findElement(By.id("login")).clear();
4 driver.findElement(By.id("login")).sendKeys("admin");
5 driver.findElement(By.id("password")).clear();
6 driver.findElement(By.id("password")).sendKeys("admin");
7 driver.findElement(By.xpath("html/body/div[1]/div[2]/div[1]/div/form/fieldset/button")).click()
      ;
8 assertTrue(driver.findElement(By.xpath("//*[@id='userProfileBox']/h3/span")).getText().contains
      ("Firstname001 Name001"));
```

Listing 2.1: E2E test script portion for Claroline Login page (Java).

The choice of proper locators is fundamental for the maintainability of the test suite. Indeed, a test suite for a modern web application has to be continuously maintained to stay aligned with the current version of the web application. Structural changes in the application (i.e. changes

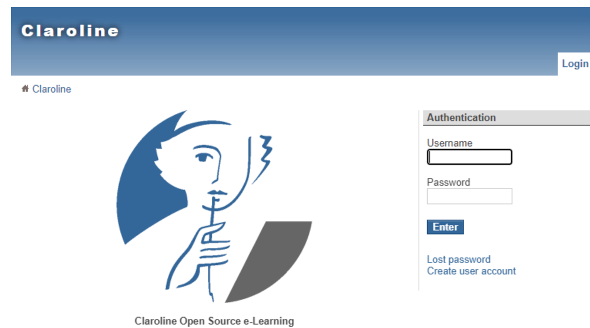

Figure 2.4: Login page of the Claroline web application

that impact the page layout and structure) will often change the DOM of the web pages, and this may require to change the locators used to interact with the page elements to reflect the new DOM structure.

A good example of a widely used locator that must be carefully selected in order to be maintainable is the XPath [LSRT14]. XPaths are expressions used to locate an element in the DOM of a Web page, and can be of two different types: absolute or relative. A relative XPath expresses the position of an element relatively to other elements of the page (e.g. the XPath `//div[@class='container']/a` selects an anchor element (`a`) inside a `div` element of class `container`). An absolute XPath instead expresses the absolute position of an element, starting from the root of the DOM. For example, `/html/body/div[1]/div[2]` is an absolute XPath that selects the second div (`div[2]`) inside the first div (`div[1]`) in the body (`body`) of the page (`html`). Both kind of XPaths can break if the DOM of the page changes, but absolute XPaths are easier to break because any change in the DOM structure affects them, while relative XPaths are affected only by changes of the near elements used to locate the target one.

For what concerns test maintainability, a test script written like the example in Listing 2.1 suffers of another serious problem that affects maintainability: the strong coupling and low cohesion problem [RLS19]. As you can see, the code in Listing 2.1 includes locators, URLs and input values hardwired as strings in the test code, i.e., the test code is *strongly coupled* with the web page structure and input data. Such coupling has a major impact on the maintainability of the test suite, since every change in the structure of the page that breaks some locators, requires every occurrence of such broken locators to be manually replaced in the whole test suite. This greatly increases the fragility of the test suite, makes the test suite prone to flakiness, and makes finding the root cause of failures harder. Low cohesion means that logical aspects (what to test) and technical aspects (how to test) are mixed: in this example, the test script has low cohesion because it contains WebDriver calls in its body, worsening the readability of the code.

A widely used solution to the strong coupling and low cohesion problem is the adoption of the Page Object pattern, that we will present in the next subsection.

2.3.1 Page Object Pattern

In this subsection, we briefly describe the Page Object pattern [POm], a design pattern that aims to reduce code duplication and improve test maintenance. It is a 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 2.2 (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 2.2: 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's 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 2.3.

```
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;
    }
```

```java
    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 2.3: Test code with page objects for our login page

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

It is important to note that, in Listing 2.3, the lines *driver.get("http://www.example.com/login");* and *driver.findElement(By.id("loginBtn")).click();* send a request to the Web server of the WAUT, and hence need to wait for the response. Without implementing any waiting strategy, this code has a high probability to fail if the page is not loaded almost instantly. In the next subsection, we will describe the different waiting strategies offered by the Selenium WebDriver framework.

2.3.2 Waiting Strategies for E2E Web Test Scripts

In this section, we provide an overview on waiting strategies used in E2E web testing, focusing on the solutions offered by the Selenium WebDriver framework. In E2E web testing the simpler mechanism to wait for asynchronous calls is the use of thread sleeps. 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 WAUT 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. [LHEM14] and Ricca and Stocco [RS21] papers. Since thread sleeps are not an optimal solution, 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*.

2.3.2.1 Explicit Waits

Explicit waits are a waiting mechanism offered by the Selenium WebDriver framework, that allow to wait for a certain condition to be verified before proceeding with the execution of the test script. Differently from thread sleeps, that wait for a fixed amount of time, explicit waits only wait until the condition is verified, making them a more efficient solution. The developer defines a maximum timeout, and if the condition is not verified until the timeout expires, the explicit wait will throw an exception that, if not caught, makes the test fail. 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;
- 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 2.4 shows an example of thread sleep from one of the test suites used to validate our tool SLEEPREPLACER, while Listing 2.5 shows the code produced by SLEEPREPLACER 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 2.4: Example of thread sleep

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

```java
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 2.5: 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 2.5), 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.

2.3.2.2 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 seconds is shown in Listing 2.6.

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

Listing 2.6: Example of implicit wait setting

When an implicit wait of 5 seconds is set, and an interaction with a page element is made, the WebDriver will poll the DOM for 5 seconds 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 the Chapter 10 of the book "Python Testing with Selenium" [Rag21].

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 [Rag21]. That is why many academic and professional sources recommend to use explicit waits as best practice in web testing.

2.3.2.3 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 2.7 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 2.7: Example of fluent wait

During the development of our tool SLEEPREPLACER, 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. For what concerns the first point, a survey published in 2020 by García et al. [GGGMO20] reported that the most used waiting strategy in Selenium WebDriver are implicit waits (62.5%), followed by explicit waits (12.5%). Fluent waits are the least used waiting strategy, used only by 4.17% of the survey's participants. For what concerns the second point, we did not find any reference in the scientific literature to support our statement, but the fact that fluent waits require the tuning of more parameters is evident from the Selenium documentation. Moreover, as it will be clearer when we will describe SLEEPREPLACER, trying different configurations for the parameters of the fluent waits would largely increase the execution time of our tool. On the other hand, using fluent waits with default polling time and no ignored exceptions is equivalent to use explicit waits.

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

2.4 Test Dependencies

In general, a test suite T is composed of multiple tests, i.e., $T = \{t_1, t_2, \ldots, t_n\}$ where the index of each test i represents the order of execution of each test, as defined by the tester. If such *original order* of execution is respected, all tests execute correctly (assuming the application under test is correct); on the other hand, if, for instance, t_j is executed before t_i, with $j > i$, and t_j fails, then t_j *depends* on t_i, and this is a *manifest dependency* [ZJW$^+$14a, GBZ18, BSM$^+$19]. In the context of web testing, a test dependency happens when a test script relies on a state of the web application created by other previously executed test scripts. For example, let us consider two test scripts for a certain web application, i.e., `AddUser` and `DeleteUser`. The first one tests the functionality of adding a user to the system, while the second exercises the delete user functionality. A convenient way to design such tests is to execute them one after the other such that the `DeleteUser` test deletes the user added by `AddUser`. However, in this way the second test relies on the state produced by the first test (i.e., the presence of a user); as a consequence, it fails if the order of execution changes.

Best practices in software testing prescribe that all test scripts in a test suite should be independent [MSW11, ZJW$^+$14a, GBZ18]. Such practice also applies to E2E testing [RS21], requiring that the result of the execution of a test script should not be affected by other tests executed before it. A test script is flaky when it non-deterministically passes or fails on the same version of the WAUT, leading to different test results in different runs. Having independent tests can prevent flakiness because, as reported by Luo et al. [LHEM14], test order dependency is one of the top three category of flakiness reported in open-source projects. If a test suite contains dependent test scripts, unpredictable failures may arise when the execution order of the test suite changes, and if dependencies are not known by the developers finding the root cause of those failures can be an hard task. As reported by Luo et al., a well-known example of dependency-induced flakiness happened with the upgrade of Java 6 to Java 7. This upgrade changed the default order that JUnit (a popular testing framework for Java) used to execute tests, resulting in many unpredicted failures in dependent test suites. Such independence among test scripts is usually achieved using methods called *test fixtures*, or Before/After methods, or Set-up/Tear-down methods. These methods are executed before (setup) or after (tear-down) the execution of a test script, to prepare the state required by a test script (before) or undo the operations performed by a test script (after), such that each test script is self-contained. Most testing frameworks, like JUnit and TestNG [UZ19, GJG15], provide such functionality (the methods are called Before/After in JUnit and Set-up/Teardown in TestNG). However, even if test fixtures are used in practice, developers often find more convenient to write dependent test scripts to both simplify the implementation and to reduce test execution times [BSM$^+$19, ZJW$^+$14a]. Indeed, executing test fixtures to make each test script independent requires additional operations, as opposed to relying on the state produced by other tests.

2.5 Test Flakiness

In this subsection, we will briefly introduce the problem of test flakiness, and explain why it particularly affects End-to-end Web testing. A test script is *flaky* when may non-deterministically pass or fail on the same version of the WAUT, i.e. leading to different results in different runs on the same AUT without any change in both the app and test code [EPCB19, ZPSH20]. The flakiness problem is very insidious for companies because: (1) it makes lose confidence in the results of the execution of the test suites with false alarms, (2) it increases deployment/release times, and generally, (3) it increases development costs [ZPSH20]. Many big companies, such as Google, Facebook and Microsoft are facing this problem [ZPSH20] and unfortunately effective solutions that allow to reveal and resolve the flakiness do not exist yet. Most of the proposed methods depend on test repetition, i.e. a test script is applied to the same WAUT for a given number of times: if the results are different, then the test is marked as flaky [LHEM14]. Test flakiness can affect all kinds of testing, but it is particularly present in E2E Web test scripts since, given the distributed and asynchronous nature of E2E Web testing, there are many possible causes than can make a test script fail. Here we present a non-exhaustive list of possible causes of non-deterministic failures in E2E Web test scripts:

- network problems that delay the loading of a page or element [MAAB+20];
- rendering problems that may lead to an incorrect visualization of an element (e.g. an element is covered by another element) [MAAB+20];
- issues related to the configuration of the machine executing tests (e.g. RAM and CPU overloading, different screen resolutions, configuration of the operating system) [MAAB+20];
- sub-optimal locators for an element, that may not be valid if other elements are displayed on the page [PMHHS19]. A classical example for this problem is the use of absolute XPaths;
- unknown dependencies between test scripts, that make the test execution fail when the execution order is changed [SLO+19, LHEM14];
- incorrect waiting for elements to be loaded before interacting with them [RSG+21, LHEM14, PHR+22].

Test dependencies and test flakiness are two intertwined problems that affect Web testing. Dependencies can become a root cause of flakiness if they are not known to the developer or not managed properly (i.e. by only running test schedules that respect test dependencies). As reported by Luo et al. [LHEM14] not respected test dependencies are one of the top three cause of test flakiness. Moreover, test flakiness can have a greater impact on the execution of a dependent test suite because if during the execution of the test suite a test script fails because of flakiness,

other test scripts that depended on the failed test will fail too, because the WAUT will not be in the state required by the dependent test scripts. Finally, test flakiness can also prevent the use of dependency detection approaches. As we will explain better in Chapter 3, different approaches for dependency detection have been proposed in scientific literature. Many of these approaches require to execute the test scripts in different orders and check their results in order to detect dependencies. But if test scripts may pass or fail non-deterministically because of flakiness (in this case, hypothetically, with other root causes rather than broken dependencies), these dependency detection approaches cannot produce correct results.

2.6 Parallel Execution of E2E Web Test Suites

Continuous integration is an established practice among web development teams, as it eases the collaboration among team members towards the delivery of the web application. The web application is built and tested continuously as code changes are made, in order to give timely feedback to developers. However, the execution time of the test suites is a bottleneck in continuous integration, especially when executing an E2E test suite, as its sequential execution can last several hours [BKMD15, GD13, Las05]. A solution to speed up the execution of E2E test suites and make them compatible with the continuous integration setting, is to parallelize the execution of the E2E test scripts, such that multiple functionalities of the WAUT can be tested on multiple browsers in parallel.

Most testing frameworks, like TestNG and JUnit, provide support for parallelizing the execution of test scripts. Such frameworks can be easily configured to run different tests in parallel, at different levels of parallelism. In particular, parallelism in TestNG is managed using two parameters, namely `thread-count` and `parallel`. The `thread-count` parameter allows the user to choose a maximum number of threads that will be used to run test scripts in parallel. For instance, if the `thread-count` parameter is set to five a maximum of five tests can be executed in parallel, while the remaining ones are put in a queue. The `parallel` parameter, instead, allows to choose the parallel execution strategy to adopt. TestNG supports four different parallel execution strategies[3], namely:

- *methods*: all test scripts (called *methods*) are executed in separate threads
- *tests*: all test scripts grouped in a “<test>” tag are executed in separate threads;
- *classes*: each test suite (called *class*) is executed in a separate thread, where its test scripts are executed sequentially;

[3]TestNG documentation, Parallelism section: https://testng.org/doc/documentation-main.html#parallel-running

- *instances*: all test scripts in the same test class instance are executed in the same thread. This option is equivalent to the previous one, unless there are multiple instances of the same test class[4].

Both TestNG and JUnit include options to run all test scripts in a test suite in parallel or all test suites in parallel. However, such frameworks do not provide a mechanism to handle multiple browsers, which is important when executing E2E test scripts in parallel. Without such feature, managing multiple browsers should be done manually by developers in the test scripts, which complicates the implementation and is prone to error.

2.6.1 Selenium Grid

Selenium Grid[5] is a framework that allows parallel execution of Selenium WebDriver test scripts. In particular, Selenium Grid manages the communication between the test scripts and multiple browsers, allowing both the parallel execution of different tests in different browsers and of the same tests on different browsers (e.g., for cross-browser testing). Selenium Grid has two core components, that can be deployed on different machines:

- *nodes*, which handle the browsers on the machines where the nodes are running. Nodes receive the commands from the hub and execute them on the browsers;
- *hub*, that is the central component of the grid. The hub receives requests to open a WebDriver session from the nodes, assigns an available browser to a session and keeps track of which node is assigned to which session. These tasks are performed by different subcomponents (i.e., Router, Distributor, Session Map, New Session Queue, Event Bus), described in more detail in the Selenium Grid documentation.

Selenium Grid can be executed in a standalone configuration (the hub and the nodes are a single entity) or in the classic hub and nodes configuration, where the hub and the nodes can be deployed both on the same machine or on different machines, even with different operating systems. In fact, since Selenium Grid is a Java software that can also be deployed in a Docker container, the same grid may contain machines running different operating systems, which allows performing cross-browser testing in ways that are not possible on a single operating system (for example the browser Safari is available only on Mac OS, while Internet Explorer was available only on Windows). When using Selenium Grid, the test scripts connect to the Selenium Grid hub, which acts

[4]Parallelization with TestNG - Accessed 2022-08-17: https://santiautomation.github.io/miscellaneous/2020/09/13/TestNG-Parallelization.html

[5]Selenium Grid official website and documentation - Accessed 2022-08-17: https://www.selenium.dev/documentation/grid/

as a central controller to assign test scripts to available nodes. If there are nodes available the test script is assigned to the first available node, otherwise the test execution is put into a queue. In an ideal scenario where a test suite does not have dependencies, Selenium Grid can be employed out-of-the-box by simply setting up the hub and nodes, and then launching the test suite using the parallel execution features offered by the employed testing framework (e.g. TestNG, JUnit). The number of test scripts running at the same time will be managed by the testing framework, their mapping with the browsers running in Grid nodes will be managed by the Selenium Grid Hub. But if the test suite does have dependencies, the developer must find a way to ensure that during the parallel execution the dependencies are respected: we will present our solution in Section 4.1.1. Docker is a containerization platform that allows to distribute applications in a portable and efficient way by using containers, that can be seen as lightweight virtual machines that include everything the contained application needs to work. In E2E software testing, Docker can be used to run multiple instances of the WAUT on the same physical machine, enabling parallel testing without interferences. The Docker-based implementation of Selenium Grid[6] offers a functionality called Dynamic Grid, that can autonomously start and stop Docker containers that run the browsers required by the tests, allowing a significant RAM saving. It is important to note, in fact, that in the standard execution mode of Selenium Grid, browsers must be running before test execution starts, and it can possibly require huge amounts of RAM depending on the number of browsers needed (that is clearly related to the number of tests to execute).

[6]https://github.com/SeleniumHQ/docker-selenium

Chapter 3

Related Work

This chapter presents an overview on existing works in scientific literature that are related to the topics of this thesis. We will present most relevant works about test dependency, test flakiness and test parallelization, and in some cases we will explain the differences between their work and our two proposed tool-based approaches, namely STILE and SLEEPREPLACER.

3.1 Test Dependencies

The problem of test dependency started to be investigated in research only in recent years: older works usually limit to prescribe test independence where required, but give no hints on how to detect test dependencies in an existing test suite. The *dependent test detection problem* has been formalized only in 2014 by Zhang et al. [ZJW$^+$14a]. They studied 96 dependent tests from 5 issue tracker systems, formulated the problem and proposed four algorithms to address it.
To give a better comprehension, we'll report Zhang's definitions of test case, test suite, test execution and dependent test detection problem.

Definition 1 (Test) *A test is a sequence of executable program statements and an oracle — a Boolean predicate that decides whether the test passes or fails.*

Definition 2 (Test suite) *A test suite T is an n-tuple (i.e., ordered sequence) of tests* $(t_1, t_2, ..., t_n)$

Definition 3 (Environment) *An environment E for the execution of a test consists of all values of global variables, files, operating system services, etc. that can be accessed by the test and program code exercised by the test case.*

Definition 4 (Test execution) *Let T be the set of all possible tests and ENV the set of all possible environments. The function exec : T x E → E represents test execution. exec maps a test t ∈ T and an environment E ∈ ENV to a (potentially updated) environment E' ∈ ENV.*

Definition 5 (Test result) *The result of a test t executed in an environment E, denoted $R(t|E)$, is defined by the test oracle and is either PASS or FAIL. The result of a test suite $T(t_1, t_2, ..., t_n)$ executed in an environment E, denoted $R((t_1, ..., t_n)|E)$, is a sequence of results $(o_1, ..., o_n)$ with $o_i \in \{$PASS, FAIL$\}$. We use $R(T|E)[t]$ to denote the result of a specific test t ∈ T.*

Definition 6 (Manifest order-dependent test) *Given a test suite T, a test t ∈ T is a manifest-order dependent test in T if ∃ two test suites S_1, S_2 ∈ permutations(T): R($S_1|E_0$)[t] ≠ R($S_2|E_0$)[t]*

Definition 7 (Dependent test detection problem) *Given a test suite T$(t_1, ..., t_n)$ and an initial environment E_0, is t ∈ T a dependent test in T?*

Definition 6 is particularly important because existing test dependency detection techniques can be divided in two different categories: techniques looking for *manifest dependencies* and techniques looking for *data-dependencies*. Techniques looking for data-dependencies search for test scripts that read and write the same shared data, and hence are potentially in conflict. These techniques analyze test scripts, statically or dynamically, searching for accesses for shared resources. The upside of this techniques is that they usually have a by far shorter execution time with respect to manifest dependency detection techniques, and that they can find all the data-dependencies in a test suite. The downside of this approach is that they may return potentially many false dependencies (i.e. dependencies that do not make the test fail if broken), since a common access to a shared resource does not necessarily imply a manifest dependency. Moreover, another downside of data-dependency detection approaches is that they are not suited for E2E Web testing, since usually test dependencies in E2E test suites are not caused by accesses to shared resources in the test code, but rather to the state of the web application under test.

Techniques looking for manifest dependencies, as per Definition 6, search for dependencies that actually make the test fail if not respected. As we will describe more in detail later, there are different approaches to do that but they all rely in executing tests in different orders to detect failures and infer the broken dependencies that lead to such failures. Differently from data-dependency detection techniques, their results are more accurate since they report only the manifest dependencies, the ones that enforce an execution order on the test suite. The major downside of this kind of approaches is that the dependent test detection problem formulated by Zhang et al. is NP-complete [ZJW+14b]: this means that we cannot find an algorithm that fully solves it in a reasonable time, but only aim to approximated solutions through heuristic algorithms. Indeed, manifest dependency detection techniques can have very high execution times even for medium-sized test suites. The high execution time is the greatest limitation that affects manifest

dependency detection techniques: some authors say that the use of such techniques to enable test parallelization is impractical. [MSd21]

To find an approximate solution to the dependent test detection problem, Zhang et al. proposed four algorithms that find a subset of all dependent tests

1. **Reversal algorithm**: given a test suite T and its expected result R, the algorithm executes T in reverse order and checks wheter the result of any test differs from the expected result;
2. **Randomized algorithm** given a test suite T and its expected result R, the algorithm executes T in random order for a fixed number of trials, and checks wheter the result of any test differs from the expected result;
3. **Exhaustive bounded algorithm** given a test suite T and its expected result R, the algorithm executes all k-permutations of T for a bounding parameter k, and checks wheter the result of any test differs from the expected result;
4. **Dependence-aware bounded algorithm** the key idea of this algorithm is to avoid permutations that cannot reveal a dependent test. Given a test suite S_1, the algorithm determines, for every resource read by a test in S_1, which tests previously wrote on that resource. If a permutation S_2 has the same relationships, then S_2 is guaranteed to pass and does not need to be run. The algorithm is bounded by a parameter k:

   - if k=1, the algorithm executes all tests in default order, detecting which tests does not access any shared resource. Then excludes those tests and executes the remaining ones in isolation: if any of them returns a result different from the expected, then it is marked as dependent test;
   - If k $\geq$ 2 the algorithm executes each test in isolation and records the fields that each test writes and reads. It uses the isolation execution result of each test as a comparison baseline. When generating all possible test permutations of length k, the algorithm checks whether all global fields that each test (in the generated permutation) may read are not written by any test executed before it. If so, all tests in the permutation must produce the same results as executed in isolation, and the algorithm can safely discard this permutation without executing it. Otherwise, the algorithm adds the generated permutation to the result set. Finally, the algorithm adds all 1-permutations to the result set to find all dependent tests that exhibit different results when executed in isolation.

These algorithms have been implemented in the open source tool DTDetector [dtd].

In 2015, Bell et al. [BKMD15] proposed an approach to detect data dependencies in test suites. In their definition, given an ordered test suite $T(t_1, ..., t_n)$ a test t_2 depends on t_1 if t_2 reads some value that was last written by t_1; a test t_3 has an anti-dependence on t_2 if it writes the same data that was last written by t_1. Their approach has been implemented by a tool, ElectricTest, that instruments the code of the system under test to track read-after-write and write-after-read accesses. In this way, all data-dependencies in a test suite can be found by running it only once, with a slowdown measured by author as 20% on average.
Note that two tests that are data-dependent may not have a manifest dependency: the definition of manifest dependency given in Definition 6 involves test result, while data-dependency looks only at accesses to shared resources.

In 2018, Gambi et al. [GBZ18] proposed Pradet, a tool that, in authors' intentions, proposes to combine the precision of DTDetector with the speed of ElectricTest. Like ElectricTest, Pradet collects information about data-dependecies in a given test suite and stores them in a **dependency graph**: a directed acyclic graph where the nodes represent tests and the edges represent the dependencies between them. Data-dependencies stored in the dependency graph are then refined to manifest dependencies through an iterative process: for each data-dependency, Pradet schedules the execution of the tests such that all dependencies, except for the target one, are satisfied. Next, it checks that tests produce the expected outcome albeit executed out-of-order. If the outcome of the tests involved in the target data dependency does not change, Pradet removes it from graph, otherwise it marks the corresponding edge as manifest dependency. This process is repeated until all the data dependencies are removed or become manifest. This approach, as the authors say, is comparable to the dependency aware bounded algorithm. A limitation of this approach, besides the high execution time, is that it requires the test suites to be free from flakiness or other non-deterministic behaviors ([GBZ18], page 9). This, however, is a limitation that affects all manifest dependency detection approaches, and not only Pradet.

In 2019, Biagiola et al. [BSM+19] presented the first test dependency detection approach for End-to-end (E2E) web test suites, implemented in a tool named TEDD. Differently from previous approaches and tools, that are mostly oriented for unit tests, TEDD can find dependencies in test suites that use Selenium WebDriver to interact with the web pages of the application under test through a browser. The approach is conceptually similar to Pradet [GBZ18] and DTDetector [ZJW+14a], although it brings significant improvements. The approach is composed by four steps and takes in input an ordered Web test suite $T(t_1, ..., t_n)$:

1. **compute** an initial test dependency graph (TDG) by analyzing values used as inputs in test cases to find potential dependencies (this is a key difference with previous approaches, and it will be explained more in detail in the next paragraph);

2. **filter** out potentially false dependencies from the initial dependency graph by using NLP techniques on test names;

3. **refine** manifest dependencies using the iterative process proposed by Gambi et al. [GBZ18] and **recover** missing dependencies by checking, for each failing test schedule where a dependency is inverted, if the failure is due to another missing dependency in the TDG. This is done by adding to the TDG a candidate dependency to every test case preceding the failing one in T and validating them;

4. **recover** dependencies from disconnected components in the TDG by executing in isolation each test represented in the TDG by an isolated node or by a node with no outgoing edges or with zero out-degree. For each failing test, the algorithm adds a candidate dependency to all preceding test according to the original order, then proceeds to validate them.

Differently from previous approaches that discover data-dependencies by tracking accesses to shared variables in tests [ZJW+14a] or in the SUT [BKMD15], TEDD infers such information statically from test cases: in the first step, the algorithm retrieves the set S of input values submitted by the test, i.e. values submitted in the fields of the web application under test. Then, the algorithm considers each test case t_f following t and searches if, in any statement of t_f, an input value contained in S is used. If so, a candidate dependency $t_f \rightarrow t$ is added to the TDG.
This approach enables TEDD to be used in the context of E2E testing for web applications, where the inspection of the system under test could be impractical. Another important improvement brought by TEDD is the recovery of missing dependencies: Pradet assumes that the initial dependency graph contains all themanifest dependencies, thus it can only remove false dependencies from the graph, not add them. TEDD, instead, can discover unexpected dependencies through the recovery process.

More recently, Alakeel [Ala22] proposed WebTestRepair, an algorithm able to repair sequences of E2E test scripts with broken dependencies. The algorithm takes as input a web application, the test suite with a correct execution order and a prioritized version of the test suite in an order that breaks test dependencies. WebTestRepair produces as output a list of test scripts that can be executed without breakages. Differently from Pradet and TEDD, WebTestRepair does not search for all the dependencies in a test suite, but only for those dependencies that break a certain sequence of test scripts, potentially reducing the execution time of the approach.

Finally, another interesting work on test dependencies elimination comes from Shi et al. [SLO+19]: their approach, implemented by a tool called iFixFlakies, finds patches for order dependent tests to make them independent.
Note that it solves a different problem with respect to the previous approaches: iFixFlakies is not a test dependency detection tool, order dependent tests must be provided as input: iFixFlakies will only suggest patches (i.e. pieces of code that set up a state accepted by the dependent test). It relies on different definitions with respect to DTDetector, Pradet and TEDD: iFixFlakies considers dependent tests as a special case of flaky tests, and classifies test cases according to the type of dependency they introduce.

3.2 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. [LHEM14]. 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. [EPCB19]: 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. Moreover, another work by Romano et al. [RSG+21] investigated flaky tests in the domain of Web applications and mobile Android applications. They analyzed 235 flaky tests from web and Android projects, to determine the most common root causes of flakiness and the most used fixing strategies. Their study confirms that, in the web testing domain, the most common cause of flakiness is the improper waiting for asynchronous calls. They also found that flakiness caused by layout differences between different platforms is more common in web tests rather than in mobile tests, and that most common fixing strategies for flaky tests are refactoring logic implementations (46%) and fixing delays (39.3%).

Our experience confirms the results by Luo et al., Eck et al. and Romano et al., in particular on the high diffusion of the Async category as main reason of flakiness problems, that 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 [ALS19, CSEV21, Shu, PHR+22]. In particular, Pei et al. [PHR+22] published in 2022 an empirical study about test flakiness in E2E test suites. Their study analyzed a dataset composed by 62 flaky tests from 26 different projects, whose flakiness was caused by improper waiting for asynchronous calls. Their study concluded that in most cases (38 out of 62) flakiness was caused by waiting a fixed amount of time before performing actions on the web page, instead of checking the state of the DOM. Although their study involves JavaScript test suites, that rely on different frameworks rather than the Selenium WebDriver framework on which our work focuses on, their conclusions can be generalized to E2E Web testing in general, regardless of the specific testing framework used. In fact, when translated to the Selenium WebDriver domain, suggesting to check the state of the DOM instead of

waiting a fixed amount of time equals to suggesting to use explicit/implicit/fluent waits instead of relying on thread sleeps. However, in another work, Presler-Marshall et al. [PMHHS19] 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 seems more reliable, they are not a silver bullet for resolving flakiness. Malm et al. [MCLE20] 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 proposed tool SLEEPREPLACER 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 our proposed tool 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. [MAAB+20] 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 etc.) in order to find the aspects that impact the most on test flakiness. Differently from our proposed approach, their technique does not aim to refactor test suites to prevent flakiness, but instead it aims to find the root causes of flakiness in test suites that are already flaky.

Some tool-based approaches to detect test flakiness has been proposed, such as DeFlaker by Bell et al. [BLH+18], 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. Differently from SLEEPREPLACER, DeFlaker does not refactor test suites to prevent flakiness, but identifies flaky tests in a test suite. Moreover, relying on code coverage rather than on test re-executions allows DeFlaker to reduce its execution time. Another tool-based approach is iFixFlakies by Shi et al. [SLO+19]. 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.

3.3 Test Parallelization

Test parallelization is a well known research topic, with a rich literature. However, few works face the problem of parallelizing E2E web test suites, a problem with peculiar characteristics, e.g., slow test case execution, complex test case logic and interaction with a web application composed of different subsystems.

For what concerns test parallelization in general, Candido et al. [CMd17] investigated the use of test parallelization in open source projects. The authors analyzed a set of 468 popular Java open source projects, and concluded that, considering only projects whose test suite took at least a minute to run, only 19.1% of them used parallel testing. The authors reported that the main reason for not using parallelization regards concurrency issues. Such issues are well-known and reported in previous works [MSd21, PKHM21]. Indeed, running a test suite in parallel may cause non-deterministic behaviours of tests (i.e., flakiness), consisting of tests passing or failing non-deterministically. This is mainly due to the fact that running test scripts in parallel introduces concurrency issues that are not present when the test suite is executed sequentially and, depending on the characteristics of the application under test and the test suite, these concurrency problems can introduce flaky behaviors.

In the literature, there are different approaches for parallel execution of test suites with dependencies. For instance, Parsa et al. [PATP16] proposed an approach to parallelize test execution subject to constraints on available resources, constraints between tests and constraints regarding the runtime of such tests. They proposed to use the Ant Colony algorithm [DDCG99] to build near-optimal schedules (from the execution time perspective) that can be executed in parallel. Differently from our work, their approach is only theoretical given that it focuses on static scheduling of test sequences rather than on their execution. Moreover, such approach does not target the parallel execution of E2E web test suites.

Another approach that studies how to automatically parallelize test suites with dependencies is the one by Mondal et al. [MSd21]. Their approach relies on the assumption that, if the test suite is divided into test classes in a meaningful way, e.g., each test class tests a different part of the system under test, dependencies should arise only between test methods in the same test class. The approach consists of three steps: in the first step the test suite is executed in parallel with a level of parallelism chosen by the user. In the second step, tests that failed in the first step are executed sequentially. Finally, if there are failed tests in the second step, the test classes to which they belong are executed sequentially. In this way, if the cause of failure is a broken dependency, such failures are fixed in the third step. Such work is related to ours but it also presents some fundamental differences. First of all, their approach is not designed for E2E web test scripts but for regression testing in general. Indeed, to validate their approach the authors used open source Java projects from the Apache foundation. Since E2E web test scripts usually have longer runtime w.r.t. unit tests, it is not guaranteed that such approach can reduce the execution time of E2E test suites. Moreover, the approach requires that the test suite has a certain structure (i.e.,

test scripts need to be meaningfully divided into test classes), while our approach does not have this constraint.

For what concerns the parallelization of E2E web test suites, Garg et al. [GD13] proposed an approach for automatically prioritizing and distributing test cases on multiple machines, relying on the functional dependency graph of a web application. In particular, they partition the test suite into test sets and associate the test sets with each module of the functional dependency graph. Differently from STILE, they rely on a test suite automatically generated from the UML activity diagrams of the WAUT. Moreover, their work is more focused on the fault detection capability of the test suite rather than on reducing its execution time.

Chapter 4

STILE: Optimized Parallelization of Dependent E2E Web Test Suites

In this chapter we will present our proposed tool-based approach for a optimized parallel execution of E2E Web test suites that have dependencies between their tests. In Section 4.1 we will describe the challenges of parallelizing E2E Web test suites and the solutions we propose to deal with them. In Section 4.2 we describe the baseline approach that we used to compare STILE with a standard, non-optimized parallel execution of the test suites. In Section 4.3 we describe our proposed approach and its implementation, and finally in Section 4.4 we provide the empirical evaluation of our approach.

4.1 Challenges of Parallelizing Dependent Test Suites

In Section 2.6 we introduced Selenium Grid, which is a framework that enables parallel execution of E2E test scripts. As we anticipated, some aspects of test parallelization are not managed directly by Selenium Grid. Indeed, the developer has still to manage (1) the dependencies between test scripts and (2) the state of the web application under test (WAUT). The first aspect is problematic because the execution order needs to be taken into account when partitioning the test scripts into test schedules. Secondly, running a dependent test suite in parallel, always requires multiple instances of the WAUT (A instance of the WAUT is a copy of the application under test, using its own URL and port). Indeed, even if the developer ensures that each test schedule respects the dependencies, executing multiple test schedules concurrently against the same instance of the WAUT may result in test failures, since the state of the application is accessed and modified by multiple tests concurrently. In this section, we will describe the key concepts we used to enable parallelization of dependent test suites.

4.1.1 Test Dependency Graph and Warranted Schedules

As anticipated in Section 3.1, many dependency detection approaches produce a test dependency graph as a result. A *test dependency graph* (or simply dependency graph) is a directed acyclic graph that represents test dependencies in a test suite. Nodes represent test scripts belonging to the test suite, while edges represent the dependencies between them. Specifically, an edge from a node t_j to a node t_i indicates that t_i must be executed before t_j. Inferring the dependencies in a web E2E test suite and the corresponding test dependency graph is a difficult and time-consuming task, especially if conducted manually. Recently, the approach implemented in the tool TEDD [BSM$^+$19] has been proposed to automate this process. Figure 4.1 shows a portion of the test dependency graph computed by TEDD. Such test suite exercises the *Claroline* web application, a web-based learning management system. The complete test suite for Claroline is

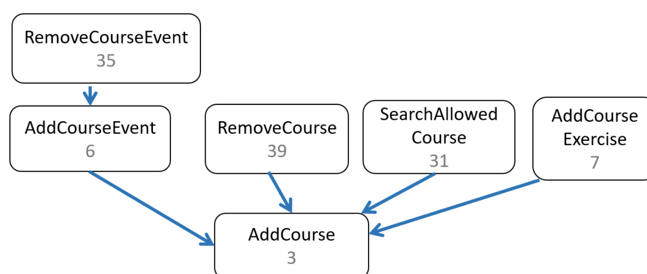

Figure 4.1: Portion of the test dependency graph generated by TEDD for the test suite of the Claroline Web application

composed by 40 test scripts. Correspondingly, its complete dependency graph is made of 40 nodes connected by 42 edges. In the Figure, the number reported in each node (below the test script name) represents the position of the test script in the original order test suite. For instance, the `RemoveCourseEvent` test is the 35th test script executed while the `AddCourse` test is the third. Since the edges represent dependencies, we can see that the `RemoveCourse` test depends on the `AddCourse` test. We can notice that, in this portion of the dependency graph, all the test scripts depend on the `AddCourse` test either directly or transitively (e.g., the `RemoveCourseEvent` that depends on the `AddCourseEvent` test which, in turn, depends on the `AddCourse` test). To implement our approach, we relied on the test dependency graphs computed by TEDD.

From the test dependency graph, it is possible to compute test schedules that respect the dependencies represented in the graph. We will call these schedules *warranted schedules*.

Definition 1 (Warranted schedule) *Warranted schedules are sequences of test scripts that respect all dependencies in the dependency graph, preserving the original order of execution of*

the test scripts. More precisely, given a test script t_i in a test suite T, its warranted schedule – warranted (t_i) – is the schedule that contains all test scripts transitively reachable from t_i in the test dependency graph, ordered as they appear in the original test suite T. By construction, these schedules are unique for each test.

For instance, in Figure 4.2, we can see which test scripts (in faded orange) belong to the warranted schedule of the `RemoveCourseEvent` test script (in orange).

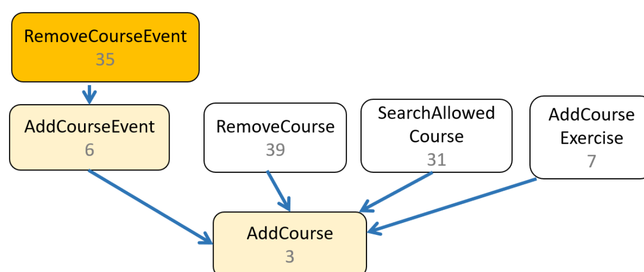

Figure 4.2: Test scripts composing the warranted schedule (in faded orange) for the `RemoveCourseEvent` test script (in orange).

The test script `RemoveCourseEvent` depends transitively on the `AddCourse` test through the direct dependence on the `AddCourseEvent` test. The remaining test scripts (i.e., those left blank), instead, depend only on the `AddCourse` test, hence they are not included in its warranted schedule. As a result, the three test scripts `AddCourse`, `AddCourseEvent` and `RemoveCourseEvent`, in this order, form the warranted test schedule of the `RemoveCourseEvent` test.

Warranted test schedules can be used for test selection and test parallelization. Regarding test selection, a test case t_i can be executed without running the whole test suite T, by running its corresponding warranted schedule, *warranted* (t_i). Concerning test parallelization, multiple warranted schedules can be executed in parallel, covering the entire test suite (i.e., executing all test scripts in the test suite). In particular, to cover the whole test suite it is enough to consider all test scripts in the test dependency graph that do not have any incoming edge (i.e., the test scripts no other test script depends on) and compute the corresponding warranted schedules.

4.1.2 Managing the Web Application State

Another issue that prevents parallel execution of dependent E2E test scripts is the shared state of the web application. In fact, if multiple test scripts are executed concurrently against the same instance of the WAUT, they share the same state, which can be polluted, causing unexpected and

unpredictable failures. We address this issue by deploying different instances of the WAUT using Docker, one for each test schedule that is executed in parallel. In particular, when using Selenium Grid, we make sure that each test schedule is executed against a different instance of the WAUT by (1) deploying each instance of the WAUT on a different URL and (2) using *parameterized* tests to pass such URLs to the respective test scripts. Parameterized tests are offered by most testing frameworks such as JUnit and TestNG. They let the testers specify parameters at runtime in order to avoid hardcoded values in the test scripts [KXL19].

4.2 Baseline Approach for Parallel Execution of Dependent Test Suites with Selenium Grid

To compare our approach with a non-optimized parallel execution of a test suite, we need a way to execute dependent test suites in parallel to be used as a baseline. We implement parallel execution of warranted schedules with TestNG and Selenium Grid using the following workflow:

1. *Generation of warranted schedules from the dependency graph*: We generate a warranted schedule for each test script with no incoming edge in the test dependency graph. In this way, we will cover the whole test suite;
2. *Generation of a TestNG XML file*: we add a "`<test>`" tag for each warranted schedule and set the parallelism level to `test`. Each "`<test>`" tag contains as parameter a URL where the WAUT instance is listening. This way, each warranted schedule is executed in parallel against a different instance of the WAUT;
3. *Setting of the `thread-count` parameter*: `thread-count` is set to the number of cores available on the machine, in order to avoid flakiness. In fact, as we empirically verified in this study, running too many tests on the same machine will inevitably produce flakiness if the machine does not have enough computation power to manage all parallel requests. To avoid this, a commonly used thumb rule is to run concurrently only as many tests as the available cores on the machine.

If there are more warranted schedules than cores, the executions of some schedules are queued. We prioritize the warranted schedules by execution time such that the schedules with the longest runtime are executed first, hence potentially reducing the overall execution time of the test suite.

Figure 4.3 shows the architecture we use for the parallel execution of test scripts with Selenium Grid. We use this approach as a baseline in our evaluation of STILE. We provision two virtual machines hosted on Amazon Web Services, one for running test scripts (Test VM) and the other for running the instances of the system under test (WAUT VM). We employ the Docker-based

version of Selenium Grid, since, in our experience, it makes easier to deploy and run browsers in Docker containers and provides better stability. As shown in Figure 4.3, the Test VM also runs the Selenium Grid hub and nodes. We made this choice to reduce costs, but if necessary Selenium Grid enables to run the test scripts and the nodes running the browsers on different machines. Instead, the Docker containers running the instances of the WAUT are deployed on a separate virtual machine (i.e., the WAUT VM), in order to reduce flakiness. Indeed, running both browsers and instances of the WAUT in the same virtual machine could exhaust its computational resources; as a consequence, some tests could become flaky since an instance is not able to process all the requests. We have experienced this problem during our tests. The launch of Docker containers, which run SUT instances, must be done before starting the test suite and is automated with a Bash script that runs before the test scripts are started.

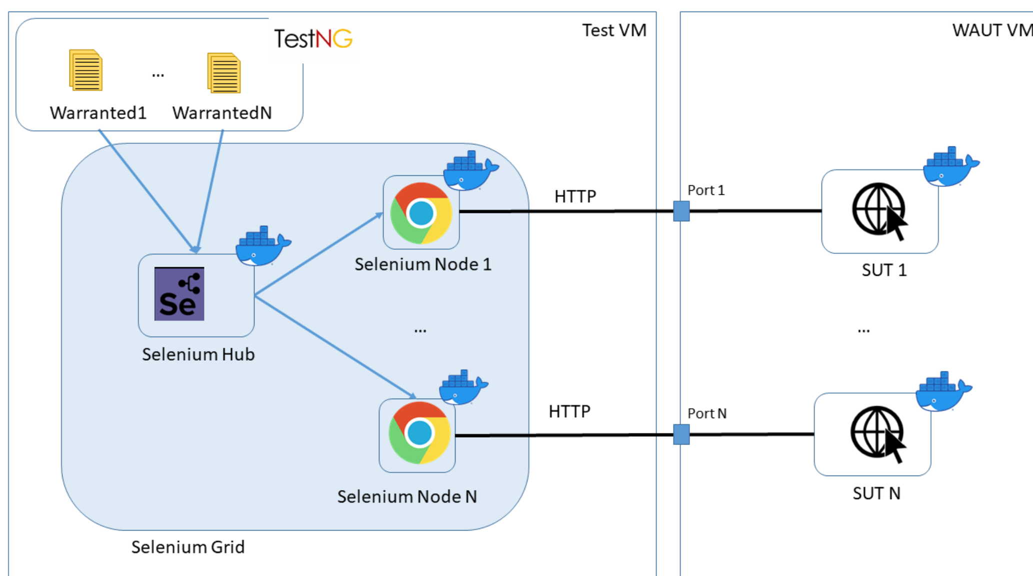

Figure 4.3: Architecture for parallel execution with Selenium Grid and TestNG

4.3 Our Proposal

Depending on the structure of the dependency graph of web the application under test (WAUT), the approach shown in Section 4.2, that runs all the warranted schedules in parallel, may not always be the most efficient solution. In fact, warranted schedules often share common prefixes composed of the same test scripts. In particular, in web applications some operations often require the execution of other operations to enable them. For instance, in our running example,

adding a course is a prerequisite for several other operations that are carried out on the added course, e.g., searching a course or removing it.

Let us consider an example of a test suite T composed of 100 test scripts (see Figure 4.4). By executing TEDD, we extract the dependency graph, which may consist of two connected components: the first including the test scripts $t_1 ... t_{50}$ and the second the test scripts $t_{51} ... t_{100}$, with, for instance, four nodes having no incoming edges (i.e., t_{49}, t_{50}, t_{99} , t_{100}). As a result,

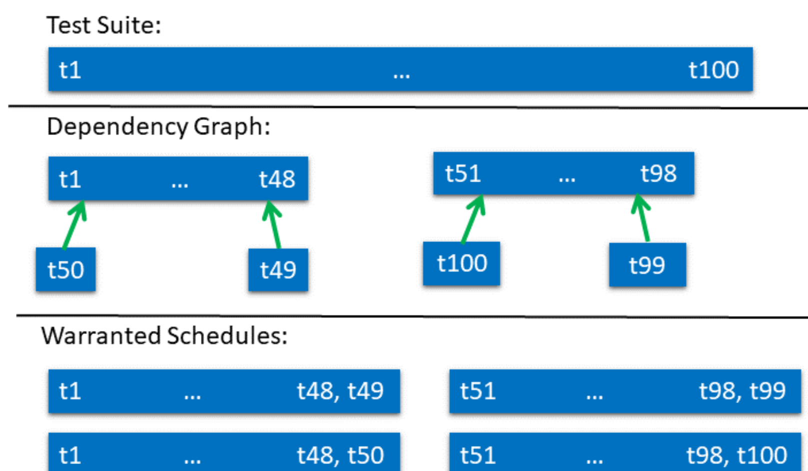

Figure 4.4: Example of test suite, its associated dependency graph and warranted schedules.

we have the four warranted schedules shown at the bottom of Figure 4.4. Assuming a fixed execution time for each test script, when running the four warranted schedules in parallel on four machines, the execution time is halved w.r.t. the sequential execution of the test suite T.

However, almost all test scripts are executed twice (except t_{49}, t_{50}, t_{99} , t_{100} which are executed once). Furthermore, if we consider a more constrained scenario of a single machine with less than four cores (i.e., less than the number of warranted schedules), the reduction in the runtime w.r.t. the sequential execution is not guaranteed. For example, with only three cores available, the execution time of the parallel execution would be equal to the sequential execution time, since one of the four warranted schedule would be left out from the initial parallel execution. Since the number of warranted schedules that can be executed on the same machine is constrained by the available computational resources, running the same test scripts in different test schedules multiple times is a non-negligible waste of resources, including computation, time and energy. The idea behind STILE is to reuse as much as possible the web application state created by the execution of a single (prefix sub-) sequence of a test script, in order that such state is reused by the other sequences that require it.

4.3.1 The Main Phases of STILE

We implemented our approach in a tool called STILE (teST suIte paralLElizer). Our approach takes as input an E2E web test suite, along with its test dependency graph, that can be either computed by TEDD or manually produced by testers (if all the dependencies in the test suite are already known). Our approach comprises four main steps, implemented by four main components, represented in Figure 4.5. Here we summarize the workflow of STILE, before describing each step in detail. STILE takes as input the test dependency graph of the target test suite and computes from it a set of warranted schedules, i.e. schedules that respect test dependencies and cover every test script in the test suite considering all the generated schedules. Then, the generated warranted schedules are represented in a prefix tree, in order to run the common prefixes only once. At this point, the actual execution of the test suite begins: the *Test Scripts Parallelizer* component visits the tree, running the test scripts and managing the instances of the application under test accordingly to the prefix tree structure. When the execution of all the test scripts is over, the *Visualizer* component produces a graphical representation of the prefix tree, with a color code to represent the execution results. The full implementation and replication package of STILE is publicly available at: https://sepl.dibris.unige.it/STILE.php.

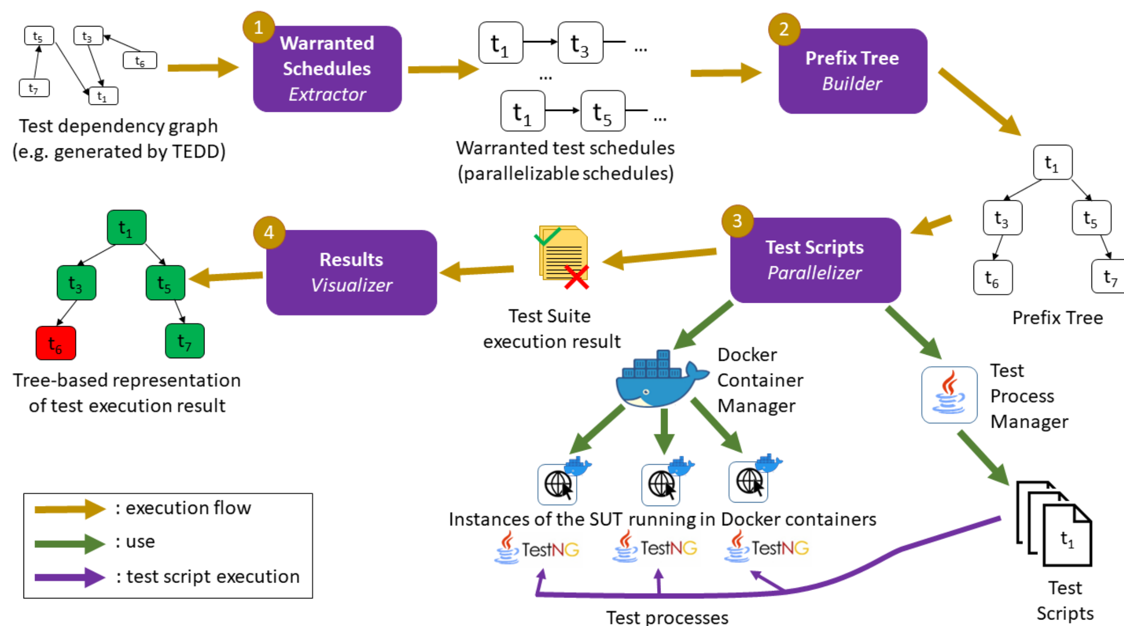

Figure 4.5: Overview of the four main phases implemented in STILE within four specific modules.

4.3.1.1 Warranted Schedules Extractor

The first step of our approach (step ❶ in Figure 4.5) takes as input the test dependency graph of the test suite and extracts the warranted schedules covering every node (i.e., test script) in the test dependency graph. To that end, the *Warranted Schedules Extractor* component generates a warranted schedule for each node with no incoming edge in the test dependency graph. For example, starting from the dependency graph shown in Figure 4.1, the extractor generates the following set of warranted schedules: {(`AddCourse`, `AddCourseExercise`), (`AddCourse`, `SearchAllowedCourse`), (`AddCourse`, `RemoveCourse`), (`AddCourse`, `AddCourseEvent`, `RemoveCourseEvent`)}. Considering the whole test suite for the web application Claroline, there are 28 warranted schedules. On average, each of the 40 test scripts is repeated two times in such schedules. The most repeated test scripts are `AddUser`, `AddCourse` and `AddMultipleUsers` respectively contained in 15, 10 and 6 warranted schedules.

4.3.1.2 Prefix Tree Builder

The second step of the approach (step ❷ in Figure 4.5) aims at reducing redundancy among the test scripts to be executed. To this end, our approach stores such schedules in a *prefix tree*. First proposed in 1959 by Renè de la Briandais [DLB59], prefix trees are data structures originally intended to store and retrieve words by character prefixes in a computationally efficient way both regarding space and time. A portion of the prefix tree for the Claroline test suite is reported in Figure 4.6 (extracted from the graph shown in Figure 4.1).

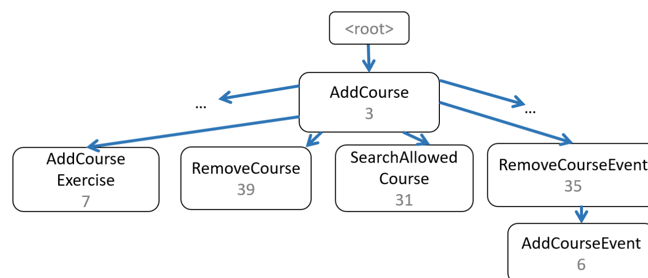

Figure 4.6: Portion of the Prefix Tree for the Claroline test suite (corresponding to the dependency graph shown in Figure 4.1)

The root of the prefix tree is a placeholder with no associated test script, while its children are the test scripts that appear as the first elements of the warranted schedules. A path from the root to a bifurcation represents a shared prefix between two or more warranted schedules, and the bifurcation represents the test script where two or more schedules differ. Note that the prefix tree may contain duplicated test scripts, depending on the topology of the test dependency graph.

In particular, a duplicated test script is present in the prefix tree if, among all the generated warranted schedules, it appears in different schedules with different prefixes. For example, in the two warranted schedules $[t_1, t_3, t_5, t_6, t_8]$ and $[t_1, t_3, t_6]$, the test script t_6 is present two times in the prefix tree, and this is unavoidable as it is preceded by different prefixes in the two schedules.

In order to build a prefix tree, the *Prefix Tree Builder* starts from the root and, for each warranted schedule, it appends all the nodes of the schedule starting from the first. If a prefix of a warranted schedule is already present in the tree, only the nodes in its suffix are appended.

4.3.1.3 Test Scripts Parallelizer

The third step of the approach (step ❸ in Figure 4.5) executes the test scripts of the prefix tree in parallel. To this end, the *Test Scripts Parallelizer* visits the prefix tree by executing Algorithm 4.1.

Algorithm 4.1: Visit algorithm used by the Test Scripts Parallelizer

```
    Input  : node – the root of the prefix tree of the test suite
             container – an instance of the WAUT
    Output: tree – the prefix tree of the test suite labeled with execution results
 1  Visit (node, container) :
 2      run test script associated to node against container
 3      n ← number of children of node
 4      if n > 1 then
 5          for i = 0; i < n-1; i++ do
 6              visit(node.children[i], container.clone())
 7          end
 8          visit(node.children[n-1], container)
 9      end
10      else if n == 1 then
11          visit(node.child, container)
12      end
13      else
14          destroy container
15          return
16      end
```

The *Test Scripts Parallelizer* oversees the execution of the test scripts and manages the lifecycle of Docker containers where the instances of the WAUT are executed. In particular, the *Test Scripts Parallelizer* module is composed by three subcomponents, not represented in Figure 4.5 for space reasons. The core component is the *Tree Parallelizer Manager*, that performs the prefix tree visit, running the tests against the proper instance of the WAUT following the previously described algorithm. When the execution is over, the *Tree Parallelizer Manager* invokes the *Results Manager* to output the test suite execution results. To complete its task, the *Tree Parallelizer Manager* relies on the following components:

- the *Container Manager* that starts, clones and stops the container encapsulating the WAUT instances;

- the *Test Process Manager* that manages the processes where test scripts are executed. At the beginning of the test suite execution, the *Test Process Manager* creates a JVM process for each node in the prefix tree. These processes then wait to receive a command from the *Tree Parallelizer Manager* with the name of the test scripts to execute as well as the URL the respective Docker container is listening on. When the execution of a test script ends, the test process returns the result (along with details about the failure reason, if any) to the *Tree Parallelizer Manager*.

The output of this step is the execution result of the test suite, represented by a labeled prefix tree that contains the execution result for each test (i.e., passed or failed) along with details about the occurred error (i.e., the stack trace), if any.

Let's see more in detail how Algorithm 4.1 performs the visit. At line 2, we execute the test script associated to the node against its container. Then we check how many children the current node has: if it has more than one children, for each child except the last a new container is created with the same state of the original container (container.clone(), line 6), then visit() is recursively and asynchronously called on each child and container. For the last child, visit is called using the same container as the current node. Otherwise, if the node has only one child (line 10), visit is called on the only child and current container (line 11). Finally, if the current node has no children, this means that it is a leaf, so the algorithm destroys the current container (line 14) and returns.

4.3.1.4 Results Visualizer

In the last step (step ❹ in Figure 4.5) the results of the parallel execution are represented in a visual format. To do that, the *Results Visualizer* component takes the labeled prefix tree, i.e., the output of the previous step, and converts it to the DOT format, assigning a different color to each node depending on the execution result of the corresponding test script (green for passed, red for failed, orange for skipped). The DOT tree is then saved in a PNG file using Graphviz [1].

4.3.2 Implementation of STILE and Deployment

We implemented STILE as a Java application that relies on Docker to manage WAUT instances and browsers. The deployment of STILE is represented in Figure 4.7. Similarly to the setup with Selenium Grid, we employ two different virtual machines for running STILE. On the first virtual machine, called Test VM in the figure, we execute the STILE core component, the test scripts and the browsers. On the second virtual machine, called WAUT VM, we execute the Docker

[1] https://graphviz.org/

containers with instances of the WAUT. Those containers are created and destroyed by STILE, that controls the Docker instance running on the WAUT VM.

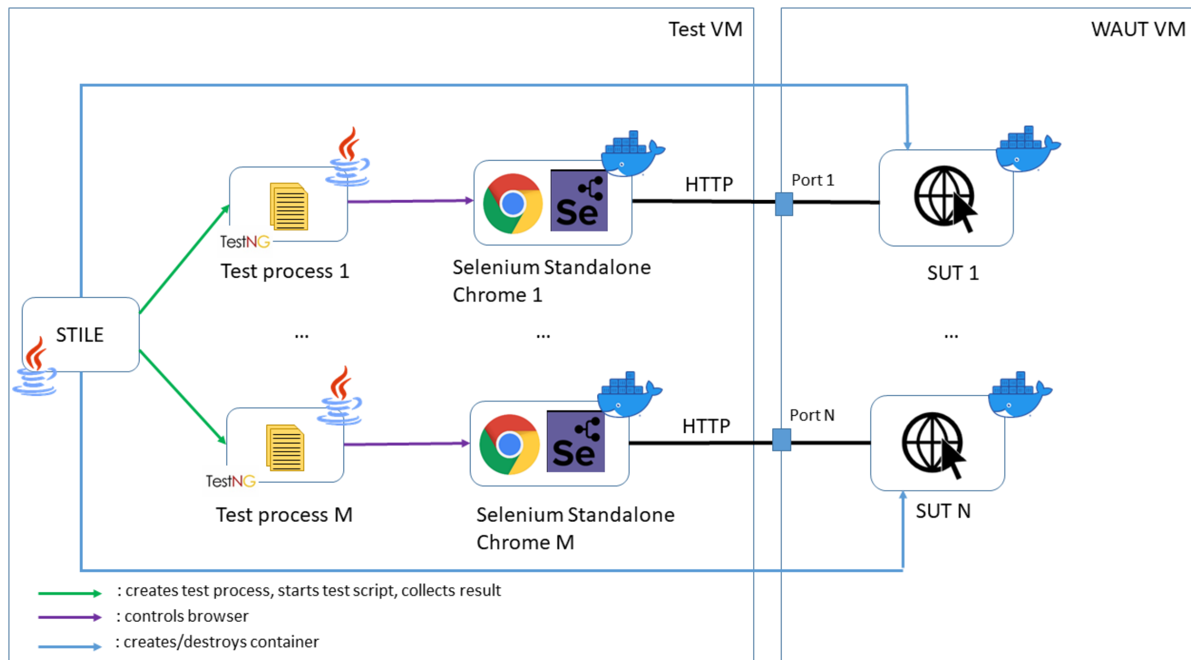

Figure 4.7: Architecture for the deployment of STILE.

Let us consider in more detail the role of the different components shown in Figure 4.7. STILE implements the steps described in the previous subsection and as well as its main components (i.e., the Warranted Schedules Extractor, the Prefix Tree Builder, the Test Scripts Parallelizer and the Results Visualizer). At the beginning of its execution, STILE starts the JVM and test processes. Indeed, we noted that starting the test processes on demand, besides increasing the overall runtime of the given test suite, introduces flakiness by requiring additional computational resources during the execution of existing test scripts. So we decided to start all the required test processes at the beginning of the execution of STILE. After being started, those processes listen to the standard input until STILE sends them two arguments, i.e., the name of the test script a certain process is requested to execute and the URL of the Docker container the WAUT is listening on.

Regarding the browser, we relied on the Docker-based Selenium Standalone Chrome; such image contains a Google Chrome browser and a Selenium Grid instance to control it, as running the browser in a Docker container gives, in our experience, more stability to the execution . We chose Google Chrome as browser since it is the most used web browser in the world [bro], although Selenium Standalone docker images are available also for Mozilla Firefox and Microsoft Edge.

The choice of using Selenium Standalone Grid to manage the browser enables a more modular execution of the tool: in fact, browsers in Selenium Standalone can be controlled by a remote machine, allowing the execution of test scripts on a different machine than the one that runs browsers. In our case, instead, we choose to locate the test processes and the browser on the same virtual machine since, for the applications we used to evaluate STILE, running tests and browsers on the same machine did not cause flakiness even at low numbers of core, so using two different virtual machines for browsers and tests would have been a useless waste of resources and money.

The WAUT VM runs only Docker containers with instances of the WAUT. No additional software is required on the WAUT VM besides Docker and the Docker image of the WAUT. Containers are created and destroyed by STILE, which communicates with the Docker application on the WAUT VM via TCP.

In Figure 4.7 we can see that STILE executes M test processes on N instances of the WAUT. The number of test processes M corresponds to the number of test scripts executions, which is equal to the number of nodes in the tree (root excluded). On the other hand, the number of instances of the WAUT N corresponds to the number of warranted schedules, i.e., the number of leaves in the prefix tree. It is important to note that M is greater or equal to the number of test scripts in the test suite since, by construction, our tree must contain every test script in the test suite. But, depending on the structure of the warranted schedules, the tree may contain duplicated tests, as already said in Section 4.3.1.2. Therefore, we have M test processes, M browsers and N WAUT instances.

The static creation of M test processes and browsers is a sub-optimal, conservative choice that we made to improve execution stability and to avoid the cost of dynamic process creation and browser startup. Indeed, the M tests that compose the prefix tree are never executed at the same time, hence it would be possible to reuse test processes and browsers to reduce RAM usage. However, it is not possible to statically determine the optimal number of test processes and browsers, equal to the maximum number of test scripts that are executed simultaneously in the entire parallel execution of the test suite, as such number depends both on the actual execution time of each test script and on how the operating system schedules the concurrent execution of the test scripts. Hence, we opted for a conservative overestimation of the M needed resources (processes and browsers), setting M equal to the total number of test scripts.

4.4 Empirical Evaluation

This section describes the design, experimental objects, research questions, metrics, procedure, results and threats to validity of the empirical study that we conducted to evaluate STILE. We followed the guidelines by Wohlin *et al.* [WRH+12] on designing and reporting empirical studies in software engineering.

4.4.1 Study Design

The focus of our evaluation of STILE is on both (a) the time saving due to the parallel execution of the test scripts, in comparison with a sequential execution of the test suite, and (b) the time and cost saving due to the prefix tree optimization of STILE, in comparison with the Selenium Grid approach that we presented in Section 4.2

The results of this study can be interpreted from multiple *perspectives*: (1) Researchers interested in an empirical comparison between different ways of parallelizing an E2E test suite; (2) Testers and Project/Quality Assurance Managers, interested in the time and cost savings that can be achieved by adopting STILE in their companies.

The *experimental objects* used to evaluate STILE are the E2E test suites of eight open-source web applications that were already used in previous studies [BSM+19, LSRT16, LRT21, LSRT18]. Table 4.1 shows relevant information concerning the test suites of the eight considered web applications. In particular it shows, for each test suite, the number of test scripts (column 2, corresponding to the number of nodes in the test dependency graph), the total Lines of Code (LOC) of the test suite (column 3), the number of edges in the test dependency graph (column 4) and the number of nodes in the prefix tree (column 5). At first glance, the test suites considered in this work may appear too simple to actually benefit from parallel execution. However, as we will see later, they require up to almost 25 minutes when executed sequentially: not long in absolute terms (e.g., in an overnight testing settings) but not negligible either. While STILE is aimed at larger industrial-grade test suites, we believe these test suites are well suited for experimentally proving its effectiveness (both in terms of cost and time).

Table 4.1: Experimental Objects.

Application / Test suite	**Test scripts**	**LOC**	**Edges in the dependency graph**	**Nodes in the prefix tree**
Addressbook	28	1411	31	37
Claroline	40	1942	42	48
Collabtive	40	2034	48	44
MantisBT	41	1830	35	41
MRBS	22	1180	21	22
PPMA	23	1304	22	26
Joomla	21	2702	22	23
ExpressCart	27	2135	25	27

4.4.2 Research Questions, Metrics, and Experimental Procedure

To evaluate the benefits of STILE in a real setting and to compare it with the sequential and the Selenium Grid (Grid, for short) executions, we used hardware resources supporting a high degree of parallelism. In particular, for analyzing the executions of the test suites on machines having different number of cores (i.e., 4, 8, 16, and 32 cores equipped with 4GB of RAM for each core) we relied on the on-demand cloud computing platform provided by Amazon Web Services[2] (AWS). All the AWS machines are equipped with the same kind of CPU running at the same frequency (i.e., 3.1 GHz). Each analysis has been executed three times to average over any random fluctuation of the execution time.

Our study aims at answering the following research questions:

RQ_1 (Time Saving w.r.t. Sequential): *What is the time saving achievable with* STILE *w.r.t. the sequential execution?*

In this RQ the comparison is with respect to the sequential execution. More specifically, to answer RQ_1, we executed each test suite with STILE and with the sequential execution, measuring the clock-time for each approach. Since the benefits from the parallel execution are more evident when machines with higher number of cores are used, we executed each test suite with the two approaches using machines with an increasing number of cores, i.e., 4, 8, 16, and 32 cores.

RQ_2 (Time Saving w.r.t. Grid): *What is the time saving achievable with* STILE *w.r.t. Grid?*

The aim of this RQ is to compare the Grid approach to parallelization with our approach. To answer RQ_2, similarly to RQ_1, we executed each test suite with STILE and with Grid and we measured the clock-time for each approach. Also in this case, we used machines with different number of cores (i.e., 4, 8, 16 and 32 cores) to analyze the difference in clock-time when increasing the available computational resources.

RQ_3 (Time Saving w.r.t. Theoretical): *What is the time saving achievable with* STILE *compared to the theoretical upper limit?*

The goal of this RQ is to analyze to which extent our approach achieves the maximum time saving (i.e., the theoretical upper limit). To answer RQ_3, we measured, for each test suite, the time required to execute the warranted test schedule with the longest runtime (measured on a machine having 4 cores). Indeed, if all the test schedules could be parallelized, the overall execution time of the test suite would be the time required to execute the test schedule with the longest runtime. Therefore, such runtime represents the maximum time saving w.r.t. the sequential execution. We compared the execution time of STILE and Grid with such runtime, i.e., the theoretical upper limit. In this way, we can quantify the parallelization overhead of STILE and Grid.

RQ_4 (Prediction of possible performance improvement): *Is it possible to predict how much*

[2]https://aws.amazon.com/

the execution time of test suites using STILE *will reduce when adding CPU cores?* The goal of this RQ is to predict how much the execution time of the test suites when using STILE will reduce when more CPU cores are available. To answer RQ_4, we compared the difference in execution time reduction (with respect to the sequential execution time) when passing from 4 to 8 cores with the difference in execution time reduction when passing from 8 to 32 cores. By using the Pearson correlation coefficient [Kir08], we found that there is a strong correlation between these two measures: this suggests that, if the execution time greatly reduces when passing from 4 to 8 cores, it will probably reduce further if we keep increasing the number of cores.

RQ_5 (Energy Consumption): *What is the reduction in energy consumption of* STILE *when compared with Grid?*

The objective of this RQ is to compare the total energy required for the execution of the test suites with STILE and Grid. When running large industrial test suites, e.g., with hundreds test scripts, the energy costs and the environmental impact are significant. To answer RQ_5, we used the CPU time as a proxy for the energy consumption. Indeed, modern CPUs drastically reduce their energy consumption when in idle state. Thus, we measured the total CPU time when executing the E2E test suites with STILE and Grid.

4.4.3 Results

RQ_1 (Execution Time w.r.t. Sequential). Table 4.2 shows the average execution time (in seconds) of the E2E test suite of each application using the sequential approach and STILE, using different machine configurations. In particular, columns 2-5 show the execution time of the sequential ("seq") run of the eight test suites employing machines having an increasing number of cores (i.e., from 4 to 32), while the remaining columns 6-9 show the time to execute the same test suites with STILE on the same machines. In the table, a red-yellow-green gradient has been used to visualize the execution times. For each test suite (row), red is associated with the highest execution time, green with the lowest and all the gradient colors represent all the other values in between. From the table, it is evident that the execution times of the sequential runs do not change significantly when considering different cores configurations (as expected by a sequential process) and are always higher than those of the STILE-based runs (the only exception is Claroline sequential executed on a 32 cores machine vs. the execution with STILE on only a 4 cores machine). It is interesting to note that, in the case of STILE, for most test suites increasing the number of cores significantly reduces the execution times.

To better appreciate this trend, Table 4.3 explicitly reports the execution time savings when considering STILE w.r.t. the sequential execution: $(ET_{seq} - ET_{STILE})/ET_{seq}$, where ET_{seq} is the time for executing the test suite sequentially and ET_{STILE} is the time for executing the test suite with STILE. For example adopting STILE with four cores allows to reduce the time required to execute Addressbook of about 62% w.r.t. the sequential execution (492s vs 1309s

Table 4.2: Time required (in seconds) to execute the E2E test suites with STILE, on different machine configurations, compared to the sequential execution.

	seq				STILE			
Test Suite	4 cores	8 cores	16 cores	32 cores	4 cores	8 cores	16 cores	32 cores
Addressbook	1309	1304	1293	1293	492	452	456	435
Claroline	1047	956	928	907	918	601	463	422
Collabtive	1397	1387	1377	1369	1139	1102	1040	1032
MantisBT	1126	1076	1032	1028	641	407	281	261
MRBS	1015	944	919	936	748	605	538	510
PPMA	718	709	705	706	492	485	483	483
Joomla	985	961	910	886	519	314	255	240
ExpressCart	1816	1779	1727	1723	581	459	433	427

respectively). Clearly the saving was measured using the same machine configuration for each of the two approaches (e.g., 16 cores for the sequential execution vs. 16 cores for STILE).

Table 4.3: Time saving (percentage) achievable with STILE thanks to the parallelization of the E2E test suites w.r.t. the sequential execution. A higher time saving implies a shorter execution time.

Test Suite	4 cores	8 cores	16 cores	32 cores
Addressbook	62%	65%	65%	66%
Claroline	12%	37%	50%	54%
Collabtive	18%	21%	24%	25%
MantisBT	43%	62%	73%	75%
MRBS	26%	36%	41%	46%
PPMA	31%	32%	31%	32%
Joomla	47%	67%	72%	73%
ExpressCart	68%	74%	75%	75%

From Table 4.3 we can see that the time saving achievable with STILE ranges from 12% to up to 75% (49%, on average). It is interesting to note that for some test suites the improvement is already significant even with a limited number of cores. For instance, in Addressbook STILE reaches a 62% saving when the test suite is executed in a machine with 4 cores (vs a 66% saving when using a 32 cores machine), Similarly, MantisBT STILE reaches a 62% saving with 8 cores

and ExpressCart 68% saving with only 4 cores. On the contrary, in other E2E test suites, the time saving increases more consistently with the number of cores. For instance, in Claroline, the time saving ranges from 12% with a 4 cores machine to 54% with a 32 cores machine; similarly in MRBS, the time saving increases from 26% to 46%. In Collabtive, PPMA, and ExpressCart the execution time seems to be relatively independent from the number of cores available, the difference in time saving from a machine with 4 cores to a machine with 32 cores being quite small (i.e., from 18% to 25% in Collabtive, from 68% to 75% in ExpressCart, while it is almost constant in PPMA).

Discussion. The execution time obtained with a parallel execution cannot be lower than the warranted schedule with the longest runtime. Since the runtime of warranted schedules depends on the structure of the dependency graph, the time saving that can be achieved by any parallelization technique (including STILE) varies in relation to such structure. In particular, if the dependency graph contains a long-lasting warranted schedule, the advantages given by parallelization are less pronounced. This is the case of Collabtive, whose test suite starts with a 18% time reduction at 4 cores, and ends with a 25% time reduction at 32 cores. Indeed, the longest warranted schedule for the Collabtive test suite is sensibly longer than the longest warranted schedules of other test suites: it takes 950 seconds, that is the 68% of the overall execution time of the test suite (1397 seconds). Such a long warranted schedule sets an upper limit to the achievable time reduction when running the test suite using STILE: in fact, since each warranted schedule is executed sequentially, the overall execution time of the test suite can never be lower than the time of the longest warranted schedule. However, the time saving cannot be predicted only by analyzing the graph structure, since the execution time of the test scripts also has a strong influence. Indeed, a warranted schedule composed of many test scripts with a low runtime may require less time than a warranted schedule with few test scripts all with a high runtime. This fact can be observed in Table 4.3, as the time saving in the execution of some test suites, does not increase consistently with the number of cores. However, the improvements obtained using STILE are always significant, ranging, in the 32 cores configuration, from 25% in the worst case (i.e., Collabtive), up to 75% in the best case (i.e., MantisBT and ExpressCart). On average, the adoption of STILE allows to halve the execution time of a test suite w.r.t the sequential execution.

In conclusion, to answer $\mathbf{RQ_1}$ we can say that the time saving achievable with STILE w.r.t. the sequential execution ranges from 12% to up to 75% (49%, on average). Adopting powerful machines (32 cores in our experiment) allows STILE to reach the highest time savings.

$\mathbf{RQ_2}$ **(Time Saving w.r.t. Grid).** Table 4.4 shows the comparison of the execution time of Grid and STILE. In detail, columns 2-5 show the time required to execute the test suites with Grid employing machines with an increasing number of cores (from 4 to 32). As preliminary observation, we can see that, differently from the case of STILE (see columns 6-9), the execution time of Grid is not always lower than the one of the sequential execution considering machine

with the same number of cores (as observed in the previous RQ for STILE, see Table 4.2). Indeed, in four cases out of eight, the execution on a 4 cores machines with Grid requires more time than the corresponding sequential execution, the reason being that Grid does not optimize the execution of the schedules, repeating the execution of common prefixes in the test schedules. Such overhead is substantial, especially on the machine configurations with less cores.

Table 4.4: Time required (in seconds) to execute the E2E test suites with Grid, on different machine configurations, compared to the parallel execution using STILE.

	GRID				STILE			
Test Suite	**4 cores**	**8 cores**	**16 cores**	**32 cores**	**4 cores**	**8 cores**	**16 cores**	**32 cores**
Addressbook	989	538	412	400	492	452	456	435
Claroline	1325	749	504	403	918	601	463	422
Collabtive	2168	1216	1006	967	1139	1102	1040	1032
MantisBT	1396	758	402	257	641	407	281	261
MRBS	1088	715	539	474	748	605	538	510
PPMA	590	438	434	432	492	485	483	483
Joomla	592	320	237	210	519	314	255	240
ExpressCart	1004	510	451	396	581	459	433	427

Focusing on the comparison between Grid (columns 2-5) and STILE (columns 6-9) we can see that, with configurations having 4 and 8 cores, the execution time of STILE requires (in several cases significantly) less time (overall in 11 out of 12 cases), while with configurations having 16 and 32 cores, the two approaches are comparable (with a slight advantage for Grid).

To better analyze this aspect, Table 4.5 shows the comparison between STILE and Grid in terms of execution time on the same hardware configurations. In particular, each cell reports the time saving related to using STILE w.r.t. Grid. We can see that, when considering a hardware configuration with a low number of cores (from 4 to 8), the time saving from the STILE adoption is substantial in almost all test suites. In particular, when considering a 4 cores machine, the time saving achieves 54% in MantisBT and an average of 36%. Similarly, STILE is still convenient with an 8 cores machine for most test suites (on average, the time saving is 13%), even though the time savings are less pronounced. When using machines with more than 8 cores, the two approaches become comparable, with a slight advantage of Grid when using a 32 cores machine.

Discussion. The results show that STILE can reduce the execution time of the test suite w.r.t. Grid when the available computational resources are limited (i.e., 8 cores or less). STILE executes much less test scripts when compared to Grid, because all the common prefixes in each warranted schedule are executed only once with STILE (this will be analyzed in detail also in Table 4.8 when answering to RQ5). On the contrary, when the computational resources exceed the number of

Table 4.5: Time saving achievable with STILE w.r.t. Grid.

Test Suite	4 cores	8 cores	16 cores	32 cores
Addressbook	50%	16%	-11%	-9%
Claroline	31%	20%	8%	-5%
Collabtive	47%	9%	-3%	-7%
MantisBT	54%	46%	30%	-1%
MRBS	31%	15%	0%	-8%
PPMA	17%	-11%	-11%	-12%
Joomla	12%	2%	-8%	-14%
ExpressCart	42%	10%	4%	-8%

warranted schedules (i.e., with 32 cores considering the complexity of eight test suites), executing less tests does not significantly impact the overall execution time, as each warranted schedule can be executed in isolation on a dedicated core. Rather, the overhead due to prefix tree optimization in STILE becomes significant and negatively affects the execution time. However, the difference between STILE and Grid at high number of cores is always very small, with an average of -8% with 32 cores.

In conclusion, to answer **RQ$_2$** we can say that the time saving achievable with STILE w.r.t. Grid is variable and depends on the machine configuration used (and the complexity of the test suite). We observed savings up to the 54% on 4 cores machine with STILE up to a slight increase in time when considering 32 cores machines.

RQ$_3$ (Time Saving w.r.t. Theoretical). Table 4.6 shows the comparison between STILE and Grid w.r.t. the execution of the warranted schedule with the longest runtime. In particular, column 2 shows the average execution time, in seconds, required to execute the complete sequential run (t_{comp}) of the eight test suites (on a 4 cores machine). Column 3 shows the average time, in seconds, required to execute the warranted schedule with the longest runtime (t_{long}) for each test suite. Thus, in column 4 we computed the theoretical maximum time saving achievable with a parallelization technique, i.e., $p_{reduction} = (t_{comp} - t_{long})/t_{comp}$ (the value is reported as a percentage). Columns 5 and 6 show the time saving achievable with STILE and Grid respectively, when both are executed on a machine with 32 cores.

Except for Addressbook, in all cases, as indeed it should be, the best time saving is the theoretical one. STILE achieves a comparable time saving than Grid in six cases out of eight while in two the savings are slightly lower (i.e., PPMA and Collabtive cases).

Table 4.6: Comparison of the time required to execute the warranted schedule with the longest runtime, w.r.t. the test suite execution with STILE and Grid using a machine with 32 cores.

Test Suite	seq			STILE 32 cores	GRID 32 cores
	complete	longest schedule			
		time	%		
Addressbook	1309	430	67%	67%	69%
Claroline	1047	358	66%	60%	61%
Collabtive	1397	950	32%	26%	31%
MantisBT	1126	184	84%	77%	77%
MRBS	1015	456	55%	50%	53%
PPMA	718	291	59%	33%	40%
Joomla	985	205	79%	76%	79%
ExpressCart	1816	385	79%	76%	78%

Discussion. The results show that STILE and Grid often achieve slightly lower time saving w.r.t. the theoretical maximum time saving. In Addressbook the maximum theoretical time saving is lower than the time saving achieved by Grid. This can be explained with the different hardware in which the warranted schedule with the longest runtime and Grid have been executed (i.e., respectively on a 4 cores and on a 32 cores machine): probably the parallelization within each test script execution, which is out of the scope of our analysis, might be slightly facilitated in the Selenium Grid framework.

Regarding the difference in time savings, in seven cases out of eight, both STILE and Grid are close to the theoretical upper bound, with a maximum difference of 7 and 6 percentage points and an average difference of 4 and 2 percentage points, respectively. Only in the case of PPMA the differences are more significant, i.e., a difference of 27 and 20 percentage points respectively for STILE and Grid. One possible explanation is that the dependency graph of PPMA has many warranted schedules that are very short (i.e., only three test scripts), diminishing the performance of STILE and Grid and hence their time saving.

In conclusion, to answer **RQ$_3$** we can say that both STILE and Grid are able to achieve time savings quite close to the theoretical maximum time saving (average difference of 4 and 2 percentage point respectively). In some cases the differences can be higher depending on the specific test suites' properties: an example is the number of leaves in the prefix tree, derived from the test suite, that strongly influences the possibility of effectively parallelizing the test suite execution.

RQ$_4$ (Prediction of possible performance improvement). In previous research questions we have shown how STILE performs against 1) the sequential execution of the test suite, 2) a parallel non-optimized execution of the test suite using Grid and 3) the theoretical limit. But previous research questions do not help in understanding if a test suite, when parallelized with STILE, will reduce its execution time if the number of cores increases.

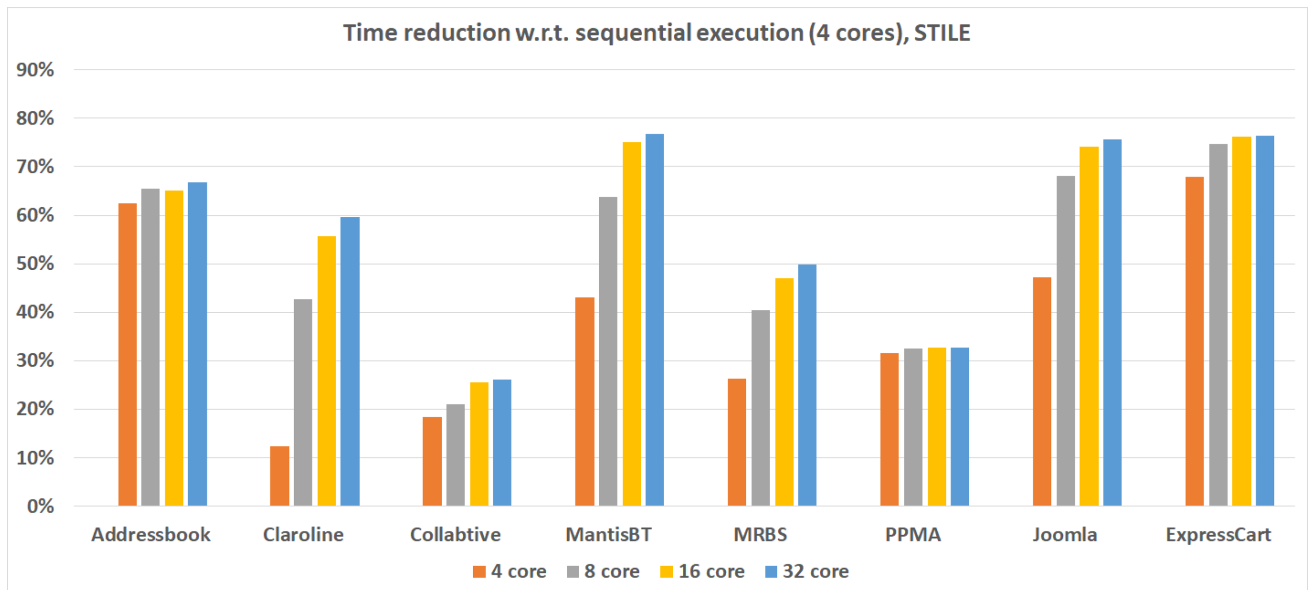

Figure 4.8: Reduction of test suite execution time when increasing the number of cores.

Figure 4.8 shows how the execution time reduction changes when increasing the number of cores. We can see that some test suites (Claroline, MantisBT, MRBS and Joomla) significantly reduce their execution time up to 16 cores, while others (Addressbook, Collabtive, PPMA and ExpressCart) approach the maximum improvement already at 4 cores, and offer a modest time reduction for higher number of cores.

The fact that a test suite does not reduce its execution time when increasing the number of cores is not necessarily a downside, it can also be an advantage: if a test suite reaches its maximum improvement at 4 cores, this means that it is not necessary to buy powerful hardware (or virtual

machines in the cloud) to execute it, since a less powerful machine will require the same time to execute the test suite. A test suite like Addressbook is better suited to be executed in parallel with STILE with respect to Claroline: the Addressbook test suite reaches a 62% time reduction at 4 cores, while for Claroline we reach a 50% time reduction only at 16 cores. But how can we tell if a test suite will have a constant improvement, or if it will reach the top at low number of cores? We do not know if it is possible to answer this question without executing the test suites, since we did not find any meaningful relation between the obtained improvement and various properties of the test suites, their dependency graphs and their prefix trees. But we found a way to answer with only two execution of the test suites, respectively using 4 cores and 8 cores: in 2023, at the moment of writing this paper, we can assume that most businesses that may be interested in using STILE for test suite execution will have at least a 8 core machine, therefore doing this experiment to evaluate the opportunity of buying powerful hardware or cloud services will not have any cost.

Table 4.7: Difference in execution time reduction from 4 to 8 cores and from 8 to 32 cores

	Difference in time reduction (4-8 core)	**Difference in time reduction (8-32 core)**
Addressbook	3.03%	1.32%
Claroline	30.29%	17.10%
Collabtive	2.67%	5.04%
MantisBT	20.76%	13.03%
MRBS	14.05%	9.42%
PPMA	0.97%	0.28%
Joomla	20.85%	7.48%
ExpressCart	6.70%	1.78%
Pearson correlation coefficient		0.924

Table 4.7 shows the difference in execution time reduction respectively from 4 to 8 cores (first column) and from 8 to 32 cores (second columns). The original values of time reductions are reported in Table 4.3. It is evident that there is a strong correlation between the achievable improvement when switching from 4 to 8 cores, and the achievable improvement when switching from 8 to 32 cores. This is also confirmed by the high Pearson correlation coefficient [Kir08], and can be observed in Figure 4.9 where the regression of the points of Table 4.7 is presented.

Discussion. The results show that different test suites show different behaviours when more

computational resources are employed. Although we cannot statically predict how much the execution time will decrease when adding more cores, we noted that test suites that benefit from more computational power (like Claroline, MantisBT and Joomla) show a high time reduction even when passing from 4 to 8 cores, and that this reduction positively correlates with the time reduction from 8 to 32 cores. Therefore, if a tester wants to know if it would be valuable to buy more resources to run a test suite using STILE, the tester may check how much the execution time decreases when passing from 4 to 8 cores: if there is a significant reduction, it may be convenient to run the test suite using higher number of cores.

In conclusion, to answer **RQ_4** we can say that if a test suite shows a great reduction in execution time when switching from 4 to 8 cores, it will likely show a great reduction in execution time when switching from 8 to 32 cores.

RQ_5 (Energy Consumption). In Table 4.8 we report the number of test scripts in the given test suites executed for each technique, i.e., sequential ("seq"), Grid and STILE. In particular, column 2 shows the number of test scripts executed when running the test suite sequentially, which corresponds to the number of test scripts in the test suite. On the contrary, when running the test suites with Grid the number of executed test scripts drastically increases (i.e., columns 3–4), since the various warranted paths share the same test scripts. Specifically, the number of test scripts executed by Grid are, on average, more than twice as much (i.e., an increase of 107%). In some specific cases such as Joomla the increment can be less drastic. This is mainly due to the length of the common prefixes of the warranted schedules: longer common prefixes result in a higher number of tests to be executed with Grid, while STILE executes these common prefixes only once. This applies both for the number of tests (Table 4.8) and for their CPU time (Table 4.9). Considering STILE (i.e., columns 5–6), we can see that the number of executed test scripts is by far lower than the test scripts executed with Grid, approaching the same number of test scripts executed with the sequential execution. Indeed, in three out of eight cases (i.e., MantisBT, MRBS, and ExpressCart), the number of test scripts executed by STILE is the same as the number of test scripts executed by the sequential execution, while in the remaining five cases, there is a slight increase, ranging from 10% to 32% (on average, the increase in the number of executed test scripts is 11%).

Column 7 shows the percentage reduction in the number of executed test scripts that can be obtained by using STILE instead of Grid. Such reduction ranges from 40% to 56% (on average, STILE requires to execute -45% test script compared to Grid).

Table 4.9 reports the CPU time required to execute each test suite using sequential (column 2), Grid (columns 3–4) and STILE (columns 5–6). From the table we can see that the CPU time increases considerably when using Grid, as the number of tests to be executed increases. In particular, the increment, w.r.t. the sequential execution, varies from 52% to 390%, with an average of 158%. On the contrary, the CPU time has a more modest increase when using STILE

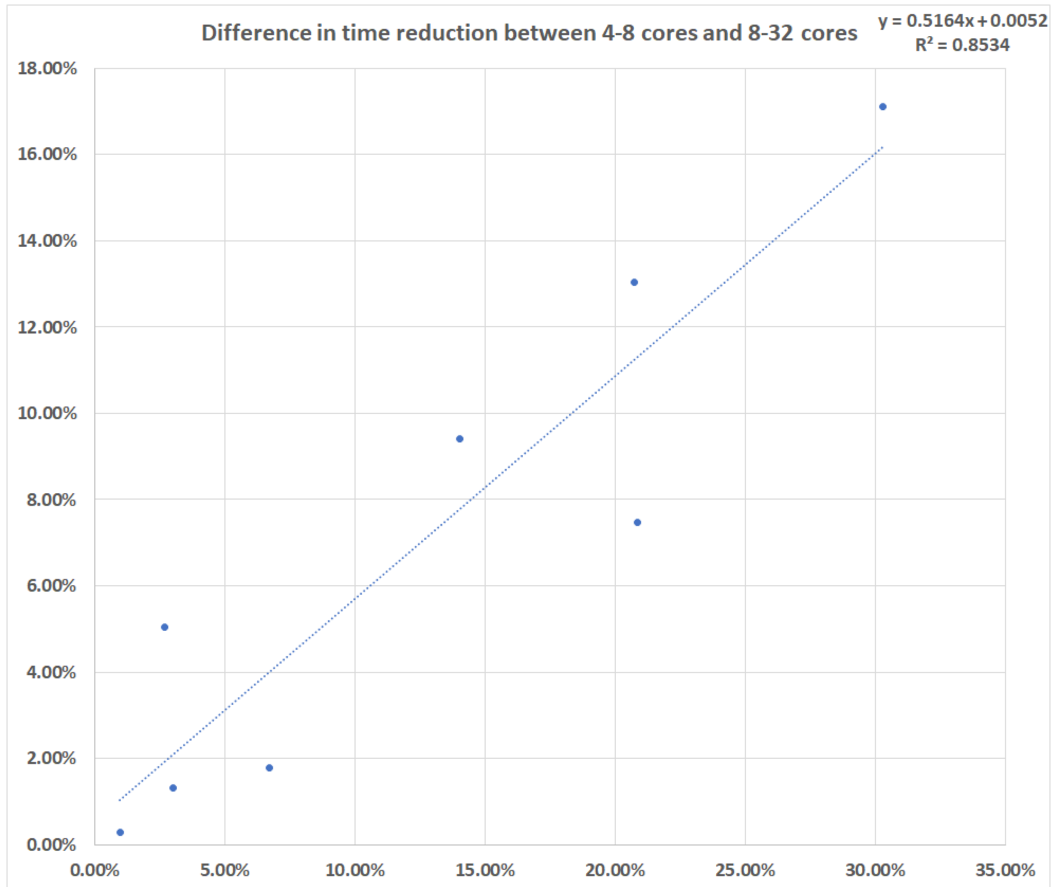

Figure 4.9: Linear regression of the differences in time reduction between 4-8 cores and 8-32 cores

w.r.t. the sequential execution. This is in line with the fact that STILE executes less test scripts than Grid while being comparable with the sequential execution. Specifically, the CPU time increment reaches a maximum of 49% while being 20% on average.

Regarding the comparison between STILE and Grid, Column 6 of Table 4.9 shows the percentage decrease in CPU time. We can see that, in all case studies, STILE uses less CPU time than Grid, with a reduction ranging from 20% to 75% with an average of 48%. Using less CPU time for executing the same test suite, implies less energy and, consequently, a lower environmental impact.

Discussion. These results show that STILE can greatly reduce the CPU time required to run a dependent test suite in parallel, with respect to the baseline approach using Grid. CPU time can be seen as a proxy for energy consumption [FWB07], although in recent years more precise metrics have been proposed [BW20, DCH10]. Even if CPU time is not a precise proxy for energy

Table 4.8: Number of test scripts executed with sequential ("seq"), Grid and STILE.

Test Suite	Tests Executed					
	seq	GRID		STILE		STILE vs GRID
	# tests	# tests	increment	# tests	increment	reduction
Addressbook	28	66	136%	37	32%	-44%
Claroline	40	81	103%	48	20%	-41%
Collabtive	40	96	140%	44	10%	-54%
MantisBT	41	94	129%	41	0%	-56%
MRBS	22	50	127%	22	0%	-56%
PPMA	23	49	113%	26	13%	-47%
Joomla	21	29	38%	23	10%	-21%
ExpressCart	27	45	67%	27	0%	-40%

Table 4.9: CPU time (in seconds) required to execute a test suite with sequential ("seq") Grid and STILE.

Test Suite	CPU-time					
	seq	GRID		STILE		STILE vs GRID
	seconds	seconds	increment	seconds	increment	reduction
Addressbook	1293	3449	167%	1801	39%	-48%
Claroline	907	2565	183%	1349	49%	-47%
Collabtive	1369	6703	390%	1681	23%	-75%
MantisBT	1028	2894	181%	1115	8%	-61%
MRBS	936	2107	125%	991	6%	-53%
PPMA	706	1388	97%	779	10%	-44%
Joomla	886	1349	52%	1075	21%	-20%
ExpressCart	1723	2883	67%	1791	4%	-38%

consumption, different studies [FWB07, MV05] found that there is a positive linear correlation between CPU utilization and energy consumption. We were not able to perform more precise measurements since we ran our experiments on the cloud, and thus we did not have access to the hardware Modern CPUs change their clock and voltage accordingly to the workload through Dynamic Frequency and Voltage Scaling (DFVS) [KGWB08], and this implies that the heavier the workload, the faster the clock goes, the more energy is used. Even without accounting for

dynamic frequency and voltage scaling, merely reducing the wall-clock time required to complete a task reduces the time that the machine running the task must stay on, therefore reducing energy consumption. In 2005, Mahesri and Vardhan [MV05] conducted an empirical study about the energy consumption of a laptop, and found that energy consumption of the CPU and other components is always higher when they are running a task than when they're idle.

In conclusion, to answer **RQ$_5$** we can say that the reduction in energy consumption of STILE when compared with Grid is significant, ranging from 20% to 75% with an average of 48%.

4.4.4 Summary of the Findings

On 32 core machines, our results suggest that STILE and Grid show comparable clock time savings, although STILE requires by far less CPU time and correspondingly uses less energy. Indeed, under the hypothesis of an execution environment with unlimited computational resources, the time required by STILE to complete the execution of a test suite is comparable to the time required to execute the warranted schedule with the longest runtime. However, in more realistic scenarios, i.e., when computational resources are limited, our experiments show that STILE can significantly reduce the time required for executing the given test suite w.r.t. Grid, by avoiding repeated executions of the same test scripts. These results are particularly interesting in a multitude of scenarios, depending on where the E2E test suite is executed:

- off premise, i.e., in a cloud computing environment. Indeed, cloud solutions often associate costs to the use of computational resources. Our results show that STILE requires less CPU time (see Table 4.9) than Grid to execute the same test suite when both approaches are executed on the same hardware configuration (i.e., on average a saving of 48%), hence reducing costs significantly;
- on premise, i.e., on a local server. In this case the computational resources are limited and cannot be easily added as in the previous scenario. Our results show that, when working with machine having a limited processing power (in relation to the needs of the test suite), STILE can reduce the execution time of a given test suite by up about the 54% w.r.t. Grid. In the industry assuming to have limited computational resources on a local server is reasonable since rarely companies invest in (or upgrade to) expensive hardware unless absolutely necessary. In that on premise scenario, differently from the off premise scenario where computational resources (i.e., cores) can be added easily, adopting STILE can help to drastically reduce the execution time w.r.t. Grid since we can consider the number of cores available as a bounded resource (adding more cores requires to upgrade/replace the hardware resources and it cannot be easily done). In this case, we can have a situation close to the one seen in the 4-8 cores cases, where STILE provides by far better results than Grid. Although our experiment does not show it, we can surmise that these good

results may scale as the size of the test suites increases. Indeed, considering complex industrial test suites, requiring more computational resources and composed by hundreds of tests, such good results could be probably obtained also with more powerful configurations (e.g., on 16-32 machines) and so STILE could be useful to reduce costs.

Moreover, both on and off premise, STILE reduces the overall energy consumption and consequently the environmental impact. When running large industrial test suites, e.g., with hundreds of test scripts, the benefits in terms of reduction of energy costs and environmental impact are significant. Indeed, the energy consumption of modern CPUs can vary drastically between the idle (low frequency) and full load (high frequencies) modes.

4.4.5 Threats to Validity

Internal validity threats concern possible confounding factors that may affect dependent variables, i.e., the time saving (RQ_1, RQ_2, RQ_3), the magnitude of the time saving per additional core (RQ_4), and the CPU time reduction (RQ_5). The main confounding factor that could have influenced our measurements is the overhead due to the execution of other processes on the machines used for the experiment. Indeed, in a physical machine there are always several background processes that are active when executing the experiments. Such processes can affect the availability of computational resources and slow down the execution of the test suites. To mitigate this threat, we make use of cloud computing resources that are dedicated to the execution of the test suites. Moreover, we execute each test suite three times, such that fluctuations in the execution time are filtered out. Another threat to internal validity are flaws in our implementation: although we are sure that there are no bugs that could lead to failure, it is surely possible that we did not code our implementation in the most efficient way.

External validity threats are related to the generalization of results. To mitigate this threat, in our empirical evaluation, six out of the eight E2E test suites that we choose have already been used in previous studies [BSM+19, LSRT16, LRT21, LSRT18] and that belong to different domains. Further experiments on additional subjects would be desirable to corroborate our findings. The AWS machines employed for the experiments cover a wide range of configurations from a small workstation (4 cores and 16GB or RAM) to a large enterprise server (32 cores - 128GB RAM). Therefore we have no reason to think that the results cannot generalize to other apps and test suites. The results obtained on the considered machines show a clear advantage of STILE on the less powerful configurations and a substantial par on the more powerful ones: we speculate that the advantages deriving from the adoption of STILE compared to Grid increase when adopting more complex test suites (thus more test cases and a more complex dependency graph). Thus, the positive results obtained in our experiments could be amplified in real contexts replicating on the 16-32 cores scale the results we observed in the 4-8 one. Clearly, a comprehensive empirical evaluation with industrial-grade complex test suite is required to confirm this conjecture.

Construct validity threats concern the relationship between theory and observation. In the context of our study such threats are related to the way the execution time of the different approaches is measured. To mitigate this threat, we collected the execution time for each approach in the same way by analyzing the execution logs. Another possible threat 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 mitigate this threat we used existing E2E test suites [BSM+19, LSRT16, LRT21, LSRT18] to evaluate STILE: these test suites were created long time before the development of STILE and contain many dependencies between the tests as they were designed to be executed sequentially. Thus, clearly, the structure of these test suites was not designed to benefit from any kind of parallelization. For this reason, they are good experimental objects to compare STILE with Grid and allowed us to analyze different kind of outcomes; for example note the variability of the results between the various test suites, some benefit a lot from the parallelization already with few CPU cores (e.g., Addressbook) while others require more computing power to have an appreciable effect in terms of execution speed (e.g., Claroline). Since the six applications of the original dataset (Addressbook, Claroline, Collabtive, MantisBT, MRBS, PPMA) are quite old, we added two modern applications (Joomla and ExpressCart) to ensure that our approach is not limited to older web applications, but can also deal with more modern interfaces.

Chapter 5

Test Flakiness Prevention: SLEEPREPLACER

In this chapter, we will present our proposed tool-based approach SLEEPREPLACER that aims to replace thread sleeps with explicit waits without introducing novel flakiness.

5.1 Motivation

In the previous chapters we introduced key concepts for E2E Web testing, focusing in particular on the parallel execution of test suites in presence of dependencies. In Sections 3.1 we introduced existing dependency detection techniques, and in Chapter 4 we presented our tool-based approach for running dependent test suites in parallel.

Manifest dependency detection approaches and STILE share a common requirement: the test suite must be free from flakiness, i.e., test scripts should always pass or fail in a deterministic way. If we use a manifest dependency detection approach such as [GBZ18] or [BSM$^+$19] on a flaky test suite the produced test dependency graph will not be accurate, because the test execution results used to build it are distorted by flakiness. If we use STILE to run a flaky test suite, failures in flaky test scripts may lead to the failure of entire branches of the prefix tree associated to the test suite, since usually if a test script fails, also other test scripts that depended on it will fail. Note that this limitation is not strictly related to STILE, but affects dependent test suites in general: the same thing will happen if there are failures in a sequential execution of a dependent test suite. However, running test scripts in parallel, particularly if the computational resources are limited, may worsen existing flakiness problems in a test suite.

For this reason, we consider flakiness as a major impediment for the parallel execution of dependent Web test suites, and in the following sections we propose a tool-based approach to prevent

it.

5.2 The Role of Waiting Strategies in E2E Test Flakiness

Regression testing is playing an increasingly important role in the industrial context [AET$^+$19], in particular in software development processes as DevOps [ZBCS16] 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.

However, for regression testing to be cost-effective, the test suite must be efficient [EE15, HDE14] and reliable. This is particularly true in the Web environment [ED19] 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. 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 [ED19].

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 [LHEM14]. 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 [LMST20].

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, namely the use of *thread sleeps* [RS21]. 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, some disadvantages, that we explained in Section 2.3.2. From several years, the Selenium testing framework is offering a better solution to synchronize test code and AUT, called `explicit waits`. 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.

For a more detailed description of the different waiting strategies employed in Selenium WebDriver E2E Web test scripts, please refer to Section 2.3.2.

5.3 Our Proposed Approach

In this section, 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 [POm] 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 in account the type of interaction performed by the test code after the thread sleep to be replaced.

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 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 [MAAB$^+$20] or fixing flaky tests exist in literature [SLO$^+$19]: 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 in Section 2.3.1, a page object is an object-oriented class that serves as an interface toward a Web page of the application under test [POm]. 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 Figure 5.1 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

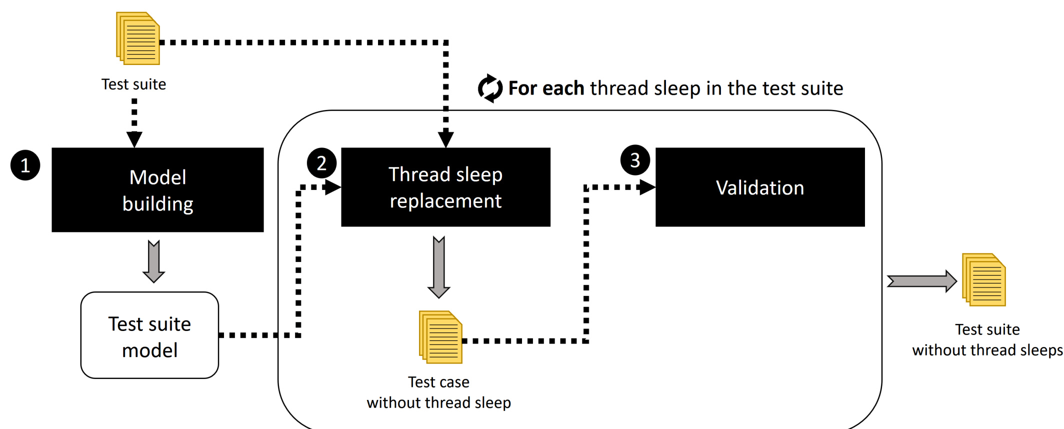

Figure 5.1: Workflow of our approach

In the next subsections, we will describe in detail the steps of our approach.

5.3.1 Step 1 - Model Building Phase

In order to automatically refactor the test suite, SLEEPREPLACER needs to build a model of the test suite, that is a object-oriented representation of the test suite structure, containing information about thread sleep locations, page access locations and type of page access performed. *Step*

Algorithm 5.1: 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 ← new TestSuite_M() // creates a new model instance representing TSuite
3      foreach test class TClass in TSuite do
4          TClass_M ← new TestClass_M() // creates a new model instance representing TClass
5          foreach test method TMethod in TClass do
6              TMethod_M ← new TestMethod_M() // creates a new model instance representing TMethod
7              add TMethod_M to TClass_M
8          end
9          add TClass_M to TSuite_M
10     end
11     // 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
12     // Case TSuite is PO-based
13     if TSuite is based on the Page Object pattern then
14         foreach page object PO in TSuite do
15             PO_M ← new PageObject_M() // creates a new model instance representing PO
16             foreach method POMethod in PO do
17                 POMethod_M ← new PageObjectMethod_M() // creates a new model instance representing POMethod
18                 usages ← all test methods that call POMethod
19                 add usages to POMethod_M
20                 foreach thread sleep TSleep in a line L of POMethod do
21                     TSleep_M ← new ThreadSleep_M() // creates a new model instance representing TSleep
22                     TSleep_M.location ← L
23                     if there is a page access PA of type PAType at line PALine after TSleep then
24                         TSleep_M.pageAccess ← new PageAccess_M(PALine, PAType);
25                     end
26                     add TSleep_M to POMethod_M
27                 end
28                 add POMethod_M to PageObject_M
29             end
30             add PageObject_M to TSuite_M
31         end
32     end
33     // Case TSuite is not PO-based
34     else
35         foreach test class TClass in TSuite do
36             foreach test method TMethod in TClass do
37                 find TMethod_M corresponding to TMethod in TSuite_M
38                 foreach thread sleep TSleep in a line L of TMethod do
39                     TSleep_M ← new ThreadSleep_M() // creates a new model instance representing TSleep
40                     TSleep_M.location ← L
41                     if there is a page access PA of type PAType at line PALine after TSleep then
42                         TSleep_M.pageAccess ← new PageAccess_M(PALine, PAType);
43                     end
44                     add TSleep_M to TMethod_M
45                 end
46             end
47         end
48     end
```

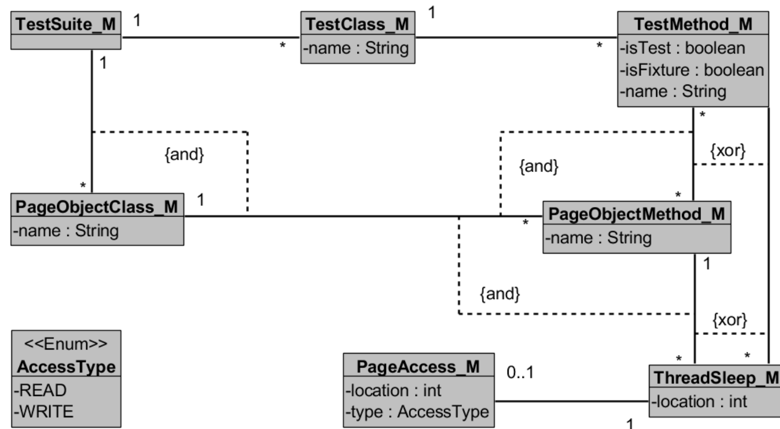

Figure 5.2: Test suite meta-model. Our tool generates, for each test suite taken in input, a model complaint with this class diagram

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 for: our tool performs the replacement relying on the information stored in the model. The procedure that implements Step 1 is described in Algorithm 5.1. 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 Figure 5.2. 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.

5.3.2 Step 2.(a) - Thread Sleep Replacement Phase

Step 2 is the fundamental one of our approach and is described in Algorithm 5.2. 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()).

Algorithm 5.2: 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

```
1  ReplaceSleep(TSuite, TSuite_M, {R}, iterations):
2      if the test suite TSuite has page objects then
3          foreach page object PO in TSuite do
4              foreach method POMethod in PO do
5                  foreach thread sleep TSleep in a line L of POMethod do
6                      find TSleep_M corresponding to TSleep in TSuite_M
7                      // 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
8                      if TSleep_M.pageAccess is not null then
9                          apply the correct rule from {R} to replace TSleep with an explicit wait
10                     end
11                     else
12                         remove TSleep
13                     end
14                     if validate(POMethod, TSuite_M, iterations) then
15                         continue
16                     end
17                     else
18                         remove the last inserted explicit wait (if any) and restore TSleep
19                     end
20                 end
21             end
22         end
23     end
24     else
25         foreach test class TClass in TSuite do
26             foreach test method TMethod in TClass do
27                 foreach thread sleep TSleep in a line L of TMethod do
28                     find TSleep_M corresponding to TSleep in TSuite_M
29                     // 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
30                     if TSleep_M.pageAccess is not null then
31                         apply the correct rule from {R} to replace TSleep with an explicit wait
32                     end
33                     else
34                         remove TSleep
35                     end
36                     if validate(TMethod, TSuite_M, iterations) then
37                         continue
38                     end
39                     else
40                         remove the last inserted explicit wait (if any) and restore TSleep
41                     end
42                 end
43             end
44         end
45     end
```

The 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 Figure 5.2), 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 Figure 5.2), 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.

5.3.3 Step 2.(b) - Validation Phase

Algorithm 5.3: 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      end
 6      else if Method is a test fixture TFixture then
 7          validationSet ← the first test method in the TClass of TFixture by analyzing TSuite_M
 8      end
 9      else if Method is a test method TMethod then
10          validationSet ← TMethod by analyzing TSuite_M
11      end
12      for i = 0; i ¡ iterations; i++ do
13          if the test suite TSuite has dependencies then
14              (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)
15          end
16          result ← run the validationSet if result == failure then
17              return false
18          end
19      end
20      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 5.3. 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 [Pal19]. 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 SLEEPREPLACER, 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.

5.3.3.1 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 5.3). 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 5.3).

5.3.3.2 Dealing with Test Dependencies during Validation

As shown in the validation Algorithm 5.3, 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 by SLEEPREPLACER 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 [BSM+19]. Our tool, when it comes to validate a dependent test t, will have to run its warranted schedule, instead of t alone, to avoid failures due to lack of previous test methods runs. Clearly, this strategy can significantly increase the execution time of SLEEPREPLACER. 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.

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. This is the same solution we employed to replicate the instances of the WAUT in STILE

5.3.3.3 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 5.3 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.

5.4 SLEEPREPLACER: Approach Implementation

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[1] 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.

5.4.1 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

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

and their use in test methods. The model is represented using Java classes, and it is an instance of the meta-model expressed in Figure 5.2. 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.
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 test method the following page access, and add its position and type to the model.

5.4.2 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 [RT01, LCRT16a, OLR21a, LRT21, LSRT16, LSRT15], and the results of our industrial previous work [OLRV21] 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 (for a more comprehensive list, please refer to Section 2.3.2.1), 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} $\rightarrow$ ExpectedConditions.visibilityOf
2. R_2 : {click, clear, sendKeys, selectByVisibleText, selectByIndex} $\rightarrow$ ExpectedConditions.elementToBeClickable
3. R_3 : {switchTo().alert()} $\rightarrow$ ExpectedConditions.alertIsPresent

We added the last rule (R_3) because, even if we did not meet them very frequently in our last refactoring work [OLRV21], 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 seconds, based on our previous experiences, and never had problems caused by a too short timeout. We did not find a reference that precisely motivates the 10 seconds timeout but, besides our experience, we think it is safe to assume that 10 seconds are more than enough time to wait for a response from the server: it is very rare that a functioning Web application, in absence of network problems, requires more than 10 seconds to reply to a request. Moreover, the 10 seconds value for the maximum time out for explicit waits is used in different examples, like the ones in the official Selenium WebDriver documentation [2] and in several books about Selenium WebDriver. [Gar22, Rag]

5.4.3 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 [Pal19].

On the contrary, the number of times a specific test method is executed depends on the presence of the PO pattern: if the test suite employs POs, the Validator will run every test method that calls the modified PO method, otherwise it will run only the test method that contains the removed thread sleep.

In Section 5.3.3.2, 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

[2] Selenium WebDriver documentation: Explicit Waits - https://www.selenium.dev/documentation/webdriver/waits/explicit-wait

[BSM+19]. 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. To summarize, 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.

5.4.4 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 5.1) 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 5.1: Code snippet from the Collabtive test suite

The first line of code in Listing 5.1 clicks on the button ”Add user” in Figure 5.3, then waits 1000 milliseconds for the form in Figure 5.4 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.

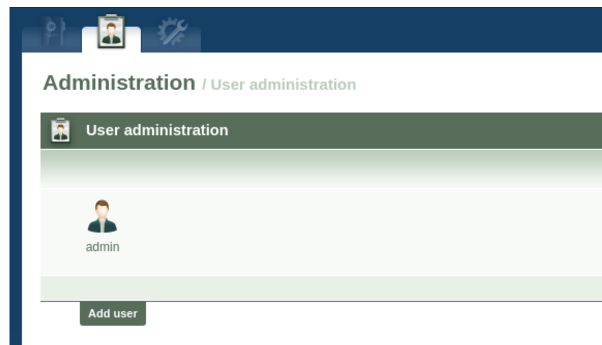

Figure 5.3: Screenshot from the Collabtive web application

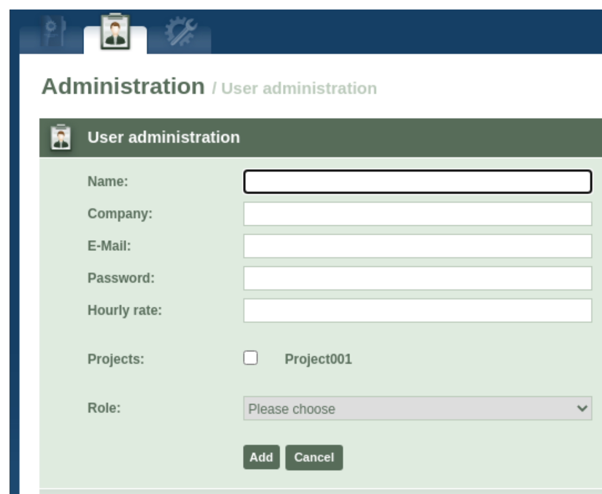

Figure 5.4: Screenshot from the Collabtive web application

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 Subsection 4.2). The resulting code is given in Listing 5.2. (note that the *wait* object has already been initialized in the Before method associated to the test).

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

Listing 5.2: 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 passes, the change is accepted and our tool moves to the next thread sleep. Otherwise, the change is reverted by restoring the thread sleep.

5.5 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.* [WRH+12] 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.

5.5.1 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 to four web applications described in the next section.

5.5.2 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, use TestNG as testing framework and Selenium WebDriver to interact with the AUT.

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

Table 5.1: Properties of the experimental objects

The PRINTO test suite has been developed in the context of a joint academia-industrial project by several junior Testers [OLRV21]. 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 [LCRT13] and then refined in further works such as [OLR$^+$21b]). 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)[3] 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[4] 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[5] is an open source web application for project management for small to medium sized businesses. It enables to manage the lifecycle of a project, that 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.

[3]PRINTO https://www.printo.it/
[4]Addressbook https://sourceforge.net/projects/php-addressbook/
[5]Collabtive https://sourceforge.net/projects/collabtive/

PPMA (PHP Password Manager)[6] 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.

5.5.3 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 to 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 Section 5.4.2. 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 STILE 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 min-

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

utes) 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.

5.5.3.1 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.

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 minutes. 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) save the state required by t to run correctly, and restore it when the tools needs 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.

5.5.4 Results

RQ1: SLEEPREPLACER Effectiveness in Replacing the Thread Sleeps

Table 5.2 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 %).

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%).

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

Table 5.2: Number of thread sleeps replaced by SLEEPREPLACER on the four considered apps

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 wait 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 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 5.3: 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 5.3: 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 5.3 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.

Application	**Total Time**	**Total**		**Replaced**	
		# thread sleeps	Time for thread sleep	# thread sleep	Time for thread sleep
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

Table 5.3: Time required (in minutes) for replacing the thread sleeps using SLEEPREPLACER

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 three minutes each). Indeed, the execution time of SLEEPREPLACER computed considering all the thread sleeps is in the range of 2.84-3.79 minutes for thread sleep; focusing only on the thread sleeps successfully replaced the time increases in the range 2.84-4.66 minutes 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 5.2).

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 minutes for thread sleep, when considering all the thread sleeps in the test suite, to 10.50 minutes 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 Section 5.5.3.1.

Thus, **to answer RQ2**, we can say that the time for successfully replace a thread sleep ranges in the interval 2.84-10.50 minutes with an average value of about 7 minutes. The actual values strongly depends by the complexity of the validation step (see Section 5.5.3.1), 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), Figure 5.1) is clearly really fast. However, we can say that the obtained execution times are 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 with three years of experience in Web testing 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 minutes on the code (i.e., 2.84 minutes per thread sleep) and (2) 1309 minutes for the validation step (i.e., 6.68 minutes 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 minutes (i.e., 9.52 minutes per thread sleep).

In Table 5.4, 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).

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

Table 5.4: Estimated Time required (in minutes) by a human Tester for executing the thread sleep replacement task

In the case of the version of the PRINTO test suite used in this study, the total estimated is of about 23h hours (1390 minutes); 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 hours (414 minutes). For the other applications the values are proportional and still relevant: about 3 hours for Collabtive and 4 hours for PPMA, while Addressbook gets

the shorter time of 28 minutes (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 minutes; while 10 minutes 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 minutes per thread sleep to replace (the most conservative estimate reported before), we have that: 307 * 3 = 931 minutes (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 minutes 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 5.5 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.

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%

Table 5.5: Effect of SLEEPREPLACER on the Test Suites execution time (in minutes)

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.

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.

5.5.4.1 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 a 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 minutes, with an average time of 7 minutes. The total times for replacing all the thread sleeps in a test suite range from 1396 minutes (approximately 23 hours, for the PRINTO test suite) to 36.6 minutes. 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 shown 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 minutes to 414 minutes. 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 even when SLEEPREPLACER is not able to replace some thread sleeps: in fact, if the validation of a explicit wait fails, the original thread sleep is restored..

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.

5.5.5 Threats to Validity

The main threats to validity affecting an empirical study are: Internal, External, Construct, and Conclusion validity [WRH+12].

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 Section 5.5.3.1, 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 generalisation 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 minimise 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).

Chapter 6

Conclusions

In this thesis, we faced the problem of test quality assurance in two different directions: 1) enabling the parallelization of E2E Web test suites in presence of dependencies and 2) preventing flakiness by using the proper waiting strategies for page elements loading.
In this chapter, we will summarize the achievements we reached and possible future works for each research direction.

1. for the first research direction, we have presented STILE, a novel approach that parallelizes the execution of E2E web test suites, minimizing their execution time while respecting their dependencies. We evaluated STILE by comparing it against the sequential execution baseline and an existing parallel strategy (i.e., Selenium Grid), using eight E2E test suites of eight open source web applications. Experimental results show that STILE is able to reduce the execution time of up to 75% w.r.t. sequential execution. When compared with the Grid implementation, STILE outperforms it when the computational resources are limited (i.e., 4 and 8 cores) while being comparable when more computational resources are available (in relation to the complexity of the test suites to execute). Moreover, our results show that STILE uses up to 54% less CPU time than the parallel execution based on Grid, enabling a reduction in energy and computation costs as well as environmental impact.

2. for the second research direction, 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 minutes to 10.50 minutes, and since this is a completely automated process, the use of SLEEPREPLACER can lead to great savings of human work.

These two research directions aim at the main goal of improving efficiency and reliability of E2E Web testing. In particular, the first research direction aims to enable test parallelization in presence of dependencies, a goal that we consider particularly important since, according to existing literature, test parallelization is heavily underused [CMd17] although it can give significant improvements in reducing the execution time of the test suites. Moreover, since E2E Web testing can be very time consuming [BKMD15, GD13, Las05] we believe it is a kind of testing that can particularly benefit from parallelization.

With the second research direction we try to prevent test flakiness, a serious problem that affects automated testing and in particular E2E Web testing. Other than the main goal of improving the stability of the execution of the test suites, with this research direction we intend to enable the use of dependency detection techniques that require to execute the test scripts and check their result, and that cannot work correctly in presence of flakiness.

6.1 Future work

In this section, we present some possible future works for the projects presented in this thesis. A future work that involves both projects is to make the implementations more generalizable, i.e. to develop tools that can support test suites written in different languages. In fact, both STILE and SLEEPREPLACER at the moment only support Java test suites that use the Selenium WebDriver framework. But in recent years, JavaScript-based testing, supported by frameworks such as Cypress[1] is becoming more and more used for testing modern web applications.

In the remainder of this section, we will now present some possible future works specific for each research direction:

1. for the first research direction, we will consider the problem of estimating the optimal number of computation units required for a given test suite execution, in order to allow further optimization of the parallel test suite execution. Moreover, we also want to try STILE with business-grade, large test suites, to see if the benefits seen with relatively small test suites can scale to higher levels. Another interesting possibility for a future work is to apply our approach to different kinds of testing other than E2E Web testing. In fact, although we explained in this thesis that the benefits of STILE (enabling test parallelization in presence of dependencies and reducing the execution time of the test suite) are particularly needed in the E2E Web testing field, the key idea behind STILE can also be applied to different

[1]Cypress: https://www.cypress.io/

kinds of testing. Finally, we want to further investigate the relation between the dependency graph of the test suite and the execution time required to run it, in order to more accurately predict such execution time. Although in Research Question 4 we give an estimated prediction of possible performance improvement, our methodology requires to run the test suite in parallel using STILE at least twice, in two different hardware configurations (4 and 8 cores). It would be useful to be able to produce an estimate of the execution time of a test suite using STILE before running it in STILE, but only looking at the properties of the test dependency graph of the test suite, its corresponding prefix tree and the sequential execution time of the test suite (that we think it is fair to assume it is already known to the developers). During the empirical evaluation of STILE, we tried to perform a multivariate regression to understand how and if properties of the test suites (i.e., the independent variables) provide a significant contribution to the time saving (i.e, the dependent variable) that can be achieved with STILE. As dependent variable, we used the angular coefficient of the regression line passing through the execution time of the test suite at different number of cores (i.e., 4, 8, 16 and 32 cores). As independent variable, we used different properties of the test suite, the test dependency graph and the prefix tree, such as the number of nodes in the prefix tree, the number of leaves in the prefix tree, the execution time of the longest warranted schedules and many others. Unfortunately, the results of the multivariate regression we performed were not statistically significant, but we hope that in the future, by using more test suites in the empirical evaluation and considering different properties of the test suites, we will be able to statically estimate the execution time of a test suite using STILE.

2. for the second research direction, 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 to calculate also the maximum timeout required by explicit waits.

Bibliography

[AET+19] Nauman Ali, Emelie Engström, Masoumeh Taromriad, Mohammad Mousavi, Nasir Mehmood Minhas, Daniel Helgesson, Sebastian Kunze, and Mahsa Varshosaz. On the search for industry-relevant regression testing research. *Empirical Software Engineering*, 24, 08 2019.

[Ala22] Ali M Alakeel. Dependency detection and repair in web application tests. *IAENG International Journal of Computer Science*, 49(2), 2022.

[ALS19] Azeem Ahmad, Ola Leifler, and Kristian Sandahl. Empirical analysis of factors and their effect on test flakiness - practitioners' perceptions. *arXiv, 1906.00673*, 2019.

[BKMD15] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. Efficient dependency detection for safe java test acceleration. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 770–781, New York, NY, USA, 2015. Association for Computing Machinery.

[BLH+18] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. Deflaker: Automatically detecting flaky tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 433–444, 2018.

[bro] Browsers market share - statcounter - accessed 2022-08-18. https://gs.statcounter.com/browser-market-share.

[BSM+19] Matteo Biagiola, Andrea Stocco, Ali Mesbah, Filippo Ricca, and Paolo Tonella. Web test dependency detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 154–164, New York, NY, USA, 2019. Association for Computing Machinery.

[BW20] Dirk Beyer and Philipp Wendler. Cpu energy meter: A tool for energy-aware algorithms engineering. In *Tools and Algorithms for the Construction and Analysis of*

Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II 26, pages 126–133. Springer, 2020.

[CMd17] Jeanderson Candido, Luis Melo, and Marcelo d'Amorim. Test suite parallelization in open-source projects: A study on its usage and impact. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, page 838–848. IEEE Press, 2017.

[Coh10] Mike Cohn. *Succeeding with agile: software development using Scrum*. Pearson Education, 2010.

[CSEV21] Bruno Camara, Marco Silva, Andre Endo, and Silvia Vergilio. On the use of test smells for prediction of flaky tests. In *Brazilian Symposium on Systematic and Automated Software Testing*, pages 46–54, 2021.

[DCH10] Georges Da Costa and Helmut Hlavacs. Methodology of measurement for energy consumption of applications. In *2010 11th IEEE/ACM International Conference on Grid Computing*, pages 290–297, 2010.

[DDCG99] Marco Dorigo, Gianni Di Caro, and Luca Maria Gambardella. Ant algorithms for discrete optimization." artificial life 5, 137-172. *Artificial Life*, 5:137–172, 04 1999.

[DLB59] Rene De La Briandais. File searching using variable length keys. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), page 295–298, New York, NY, USA, 1959. Association for Computing Machinery.

[dtd] Dtdetector. https://github.com/winglam/dtdetector.

[ED19] Ravi Eda and Hyunsook Do. An efficient regression testing approach for php web applications using test selection and reusable constraints. *Software Quality Journal*, 27:1383–1417, 12 2019.

[EE15] Edward Dunn Ekelund and Emelie Engström. Efficient regression testing based on test history: An industrial evaluation. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 449–457, 2015.

[EPCB19] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 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, page 830–840, New York, NY, USA, 2019. Association for Computing Machinery.

[FWB07] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, page 13–23, New York, NY, USA, 2007. Association for Computing Machinery.

[G+05] Jesse James Garrett et al. Ajax: A new approach to web applications. 2005.

[Gar22] B. Garcia. *Hands-on Selenium WebDriver with Java: A Deep Dive Into the Development of End-to-end Tests*. O'reilly Media, 2022.

[GBZ18] Alessio Gambi, Jonathan Bell, and Andreas Zeller. Practical test dependency detection. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–11. IEEE, 2018.

[GD13] Deepak Garg and Amitava Datta. Parallel execution of prioritized test cases for regression testing of web applications. In *Proceedings of the Thirty-Sixth Australasian Computer Science Conference - Volume 135*, ACSC '13, page 61–68, AUS, 2013. Australian Computer Society, Inc.

[GGGMO20] Boni García, Micael Gallego, Francisco Gortázar, and Mario Munoz-Organero. A survey of the selenium ecosystem. *Electronics*, 9:1067, 06 2020.

[GJG15] Satish Gojare, Rahul Joshi, and Dhanashree Gaigaware. Analysis and design of selenium webdriver automation testing framework. *Procedia Computer Science*, 50:341–346, 2015. Big Data, Cloud and Computing Challenges.

[GSHM15] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 223–233, New York, NY, USA, 2015. Association for Computing Machinery.

[HDE14] Md. Hossain, Hyunsook Do, and Ravi Eda. Regression testing for web applications using reusable constraint values. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 312–321, 2014.

[IEE94] Ieee guide for software verification and validation plans. *IEEE Std 1059-1993*, pages 1–87, 1994.

[IEE10] Iso/iec/ieee international standard - systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418, Dec 2010.

[KGWB08] Wonyoung Kim, Meeta S Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *2008*

IEEE 14th International Symposium on High Performance Computer Architecture, pages 123–134. IEEE, 2008.

[Kir08] Wilhelm Kirch, editor. *Pearson's Correlation Coefficient*, pages 1090–1091. Springer Netherlands, Dordrecht, 2008.

[KXL19] Pavneet Singh Kochhar, Xin Xia, and David Lo. Practitioners' views on good software testing practices. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 61–70. IEEE, 2019.

[Las05] Alexey Lastovetsky. Parallel testing of distributed software. *Information and Software Technology*, 47(10):657–662, 2005.

[LCRT13] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Capture-Replay vs. Programmable Web Testing: An Empirical Assessment during Test Case Evolution. In *Proceedings of 20th Working Conference on Reverse Engineering (WCRE 2013)*, pages 272–281. IEEE, 2013.

[LCRT16a] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Approaches and tools for automated End-to-End Web testing. *Advances in Computers*, 101:193–237, 2016.

[LCRT16b] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Chapter five - approaches and tools for automated end-to-end web testing. volume 101 of *Advances in Computers*, pages 193–237. Elsevier, 2016.

[LHEM14] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 643–653, New York, NY, USA, 2014. Association for Computing Machinery.

[LMST20] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. A study on the lifecycle of flaky tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1471–1482, New York, NY, USA, 2020. Association for Computing Machinery.

[LRSM22] Maurizio Leotta, Filippo Ricca, Simone Stoppa, and Alessandro Marchetto. Is nlp-based test automation cheaper than programmable and capture &replay? In Antonio Vallecillo, Joost Visser, and Ricardo Pérez-Castillo, editors, *Quality of Information and Communications Technology*, pages 77–92, Cham, 2022. Springer International Publishing.

[LRT21] Maurizio Leotta, Filippo Ricca, and Paolo Tonella. SIDEREAL: Statistical adaptive generation of robust locators for End-to-End Web testing. *Journal of Software: Testing, Verification and Reliability (STVR)*, 2021.

[LSRT14] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Reducing web test cases aging by means of robust xpath locators. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 449–454. IEEE, 2014.

[LSRT15] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Using multi-locators to increase the robustness of web test cases. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015.

[LSRT16] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. ROBULA+: An algorithm for generating robust XPath locators for Web testing. *Journal of Software: Evolution and Process (JSEP)*, 28(3):177–204, 2016.

[LSRT18] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. PESTO: Automated migration of DOM-based Web tests towards the visual approach. *Journal of Software: Testing, Verification and Reliability (STVR)*, 28(4):e1665, 2018.

[MAAB+20] Jesus Moran, Cristian Augusto Alonso, Antonia Bertolino, Claudio de la Riva, and Javier Tuya. Flakyloc: Flakiness localization for reliable test suites in web applications. *Journal of Web Engineering*, 06 2020.

[MCLE20] Jean Malm, Adnan Causevic, Björn Lisper, and Sigrid Eldh. Automated analysis of flakiness-mitigating delays. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, AST '20, page 81–84, New York, NY, USA, 2020. Association for Computing Machinery.

[MSd21] Shouvick Mondal, Denini Silva, and Marcelo d'Amorim. Soundy automated parallelization of test execution. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 309–319, 2021.

[MSW11] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. Finding bugs by isolating unit tests. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 496–499, New York, NY, USA, 2011. Association for Computing Machinery.

[MV05] Aqeel Mahesri and Vibhore Vardhan. Power consumption breakdown on a modern laptop. In Babak Falsafi and T. N. VijayKumar, editors, *Power-Aware Computer Systems*, pages 165–180, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[OLR21a] Dario Olianas, Maurizio Leotta, and Filippo Ricca. MATTER: A tool for generating end-to-end IoT test scripts. *Software Quality Journal*, pages 1–35, 08 2021.

[OLR+21b] Dario Olianas, Maurizio Leotta, Filippo Ricca, Matteo Biagiola, and Paolo Tonella. 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)*, pages 460–465. IEEE, 2021.

[OLR22] Dario Olianas, Maurizio Leotta, and Filippo Ricca. Sleepreplacer: a novel tool-based approach for replacing thread sleeps in selenium webdriver test code. *Software Quality Journal*, pages 1–33, 2022.

[OLRV21] Dario Olianas, Maurizio Leotta, Filippo Ricca, and Luca Villa. Reducing flakiness in End-to-End test suites: An experience report. In Ana C. R. Paiva, Ana Rosa Cavalli, Paula Ventura Martins, and Ricardo Pérez-Castillo, editors, *Proceedings of 14th International Conference on the Quality of Information and Communications Technology (QUATIC 2021)*, volume 1439 of *CCIS*, pages 3–17. Springer, 2021.

[Pal19] Fabio Palomba. Flaky tests: Problems, solutions, and challenges. In *BENEVOL*, 2019.

[PATP16] Masoumeh Parsa, Adnan Ashraf, Dragos Truscan, and Ivan Porres. On optimization of test parallelization with constraints. In Wolf Zimmermann, Lukas Alperowitz, Bernd Brügge, Jörn Fahsel, Andrea Herrmann, Anne Hoffmann, Andreas Krall, Dieter Landes, Horst Lichter, Dirk Riehle, Ina Schaefer, Constantin Scheuermann, Alexander Schlaefer, Sibylle Schupp, Andreas Seitz, Andreas Steffens, André Stollenwerk, and Rüdiger Weißbach, editors, *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Wien, 23.-26. Februar 2016*, volume 1559 of *CEUR Workshop Proceedings*, pages 164–171. CEUR-WS.org, 2016.

[PHR+22] Yu Pei, Sarra Habchi, Renaud Rwemalika, Jeongju Sohn, and Mike Papadakis. An empirical study of async wait flakiness in front-end testing. 2022.

[PKHM21] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–74, 2021.

[PMHHS19] Kai Presler-Marshall, Eric Horton, Sarah Heckman, and Kathryn T. Stolee. 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, page 7–13. IEEE Press, 2019.

[POm] Page Object Model.

[Rag] Sujay Raghavendra. Python testing with selenium.

[Rag21] Sujay Raghavendra. *Waits*, pages 129–142. Apress, Berkeley, CA, 2021.

[RLS19] Filippo Ricca, Maurizio Leotta, and Andrea Stocco. Chapter three - three open problems in the context of e2e web testing and a vision: Neonate. volume 113 of *Advances in Computers*, pages 89–133. Elsevier, 2019.

[RS21] Filippo Ricca and Andrea Stocco. Web test automation: Insights from the grey literature. In *Proceedings of the 47th International Conference on Current Trends in Theory and Practice of Computer Science*, 1 2021.

[RSG+21] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. An empirical analysis of ui-based flaky tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1585–1597. IEEE, 2021.

[RT01] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 25–34, 2001.

[SA20] Gian Luca Scoccia and Marco Autili. Web frameworks for desktop apps: an exploratory study. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, 2020.

[Shu] Shashank Shukla. *The Protractor Handbook*. Springer.

[SLO+19] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 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*, pages 545–555, 2019.

[UZ19] Mubarak Albarka Umar and Chen Zhanfang. A study of automated software testing: Automation tools and frameworks. *International Journal of Computer Science Engineering (IJCSE)*, 6:217–225, 2019.

[WRH+12] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[ZBCS16] Liming Zhu, Len Bass, and George Champlin-Scharff. Devops and its practices. *IEEE Software*, 33(3):32–34, 2016.

[ZJW+14a] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 385–396, New York, NY, USA, 2014. Association for Computing Machinery.

[ZJW+14b] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. Empirically revisiting the test independence assumption. technical report. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 385–396, New York, NY, USA, 2014. Association for Computing Machinery.

[ZPSH20] B. Zolfaghari, Reza M. Parizi, Gautam Srivastava, and Yoseph Hailemariam. Root causing, detecting, and fixing flaky tests: State of the art and future roadmap. *Software: Practice and Experience*, 51:851 – 867, 2020.