The Journal of Systems and Software 222 (2025) 112304

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

# STILE: A tool for optimizing E2E web test scripts parallelization ☆

Dario Olianas [a], Maurizio Leotta [a,*], Filippo Ricca [a], Matteo Biagiola [b], Paolo Tonella [b]

[a] Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Genova, Italy
[b] Software Institute, Università della Svizzera italiana, Switzerland

ARTICLE INFO

*Keywords:*
End-to-end testing
Web testing
Parallel testing
Dependency graph
Selenium
Docker

ABSTRACT

Web applications quality is commonly assessed by executing End-to-End (E2E) test scripts interacting with those systems as a human tester would. To avoid setting up the web application state for each test script, testers usually create test scripts that may depend on others previously executed. However, the presence of dependencies prevents parallelization, a fundamental technique for speedup the execution of large test suites.

In this paper, we present STILE, a tool for parallelizing the execution of E2E web test scripts that generates and executes a set of test schedules satisfying 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.

We empirically evaluated STILE on eight E2E test suites by comparing the execution time of STILE both with the sequential execution and with the parallel execution based on Selenium Grid. Our results show that STILE can reduce the execution time up to 80% w.r.t. the sequential execution and up to 50% w.r.t. Grid. Moreover, STILE provides a reduction in the CPUs usage (i.e., overall CPU-time) up to 75%.

## 1. Introduction

Ensuring the quality of web applications is of paramount importance, as ever more business-critical services are provided through web applications. 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 web application as a whole to ensure that the system behaves as expected. Frameworks implementing this approach are often adopted by practitioners since they automate the steps that a human tester would perform, i.e., interacting with the web application under test through the browser (Leotta et al., 2023).

Although E2E test scripts are used in practice, they usually have a longer execution time w.r.t. unit and integration tests, because they need to interact with the browser's graphical user interface. The front-tend part of the web application communicates with the back-end and all the third-party services that contribute to the correct functioning of the application (Radziwill and Freeman, 2020) and thus leads to the high execution time. Given their high runtime, it is difficult to integrate E2E test suites in a continuous integration setting that, instead, requires fast test cycles.

A way to speed up the execution of an E2E test suite is to run *schedules* (i.e., sequences) of its test scripts in parallel, and eventually combine the results (Bell et al., 2015), in order to reduce execution time by exploiting all available computing resources. However, parallel execution can be hindered by the fact that E2E test scripts are often dependent (Zhang et al., 2014; Ricca and Stocco, 2021). A test script dependency exists when the outcome of one test script depends on the preceding execution of another test script. In other words, the dependent test script relies on an application state which is set by the execution of one or more preceding test scripts. As a consequence, partitioning a test suite with dependent test scripts into different parallelizable schedules becomes difficult since the order relations that exist among the dependent tests must be taken into account. E2E test suites are particularly prone to dependencies as each test script exercises complex business logic scenarios requiring a specific application state (for example, the existence of a user or the availability of a product). As a consequence, enforcing isolation among test scripts is challenging as testers might rely on the application state set by previous tests when designing a test script (Zhang et al., 2014; Ricca and Stocco, 2021). Therefore, if test scripts are not executed in a specific order, test failures may happen, due to application states not properly set by the executions of previous tests.

In this paper, we present and evaluate STILE (te**ST** su**I**te para**L**lelizE**r**), a novel tool able to optimize the parallel execution of an E2E Web test

☆ Editor: Dr. Burak Turhan.

* Corresponding author.
*E-mail addresses:* dario.olianas@dibris.unige.it (D. Olianas), maurizio.leotta@unige.it (M. Leotta), filippo.ricca@unige.it (F. Ricca), matteo.biagiola@usi.ch (M. Biagiola), paolo.tonella@usi.ch (P. Tonella).

https://doi.org/10.1016/j.jss.2024.112304
Received 11 September 2023; Received in revised form 4 November 2024; Accepted 25 November 2024
Available online 9 December 2024

suite taking into account dependencies among test scripts. In order to parallelize a test suite that contains dependent tests, STILE relies on a dependency graph, i.e., a graph structure where nodes represent test scripts and edges represent dependencies between them. The dependency graph can be manually maintained by the testers during the development of the test suite, or it can be automatically computed by a dependency detection tool. In the literature there are two main categories of test dependency detection techniques: techniques looking for manifest dependencies (Gambi et al., 2018; Biagiola et al., 2019; Zhang et al., 2014) and techniques looking for data dependencies (Bell et al., 2015). Techniques looking for manifest dependencies execute tests many times in different orders, using different strategies, to find dependent tests. This kind of technique gives accurate results but is time consuming. Its execution time can be high even for medium-sized test suites, and it becomes prohibitive for very large test suites. The other kind of technique look for data dependencies between tests, by analyzing accesses the tests make to shared resources.

For this work, we rely on TEDD (Biagiola et al., 2019) to create the test dependency graph. TEDD is the first tool for manifest dependency detection especially designed for end-to-end web test suites, and it uses a combination of the two kinds of approaches described before: first it statically extracts an initial graph of potential dependencies and filters out potentially false ones, then it validates them by running tests out of order to see if they actually fail. This allows reducing the execution time usually required by manifest dependency detection tools (since the initial set of potential dependencies is reduced) while maintaining a high accuracy.

From the dependency graph, STILE infers a set of schedules that completely covers the initial E2E test suite and at the same time respects all its dependencies. Then, STILE uses a prefix tree (De La Briandais, 1959) to optimize the parallel execution of the test suite by avoiding the repeated execution of common prefixes, i.e., test scripts that are shared among different schedules. This is done by running the common prefixes only once, saving a snapshot of the state of the web application under test (WAUT), and replicating it on different instances of the WAUT. Finally, STILE manages the execution of all the schedules inside a predefined number of Docker containers using a complex run-time architecture. The novelty of this work resides in (1) the prefix tree-based approach that allows us to run a full test suite in parallel, while respecting all the dependencies and reducing as much as possible the repeated execution of test scripts; (2) the idea of replicating the state of the system under test as a way to deal with test dependencies and reduce the number of required test script executions; and, (3) the empirical validation of our tool's implementation on 8 test suites.

We empirically compared STILE against the sequential execution strategy of a given E2E test suite as well as with a reference parallel execution strategy based on Selenium Grid (Selenium, 2022), the state of the practice framework for parallel execution of test scripts, using eight existing E2E test suites for eight open source web applications. Our experimental results show that STILE is able to reduce the execution time w.r.t. the sequential execution strategy by up to 80% and the CPU-time up to 75%. In comparison to Selenium Grid, the execution time reduction is up to 50% on 4 or 8 cores, while a comparable execution time is achieved on 16 or 32 core machines, where STILE offers the significant advantage of a lower CPU usage time, which translates into lower CPU-time and potentially lower cloud computing costs.

This work extends our previous tool paper (Olianas et al., 2021) in four main directions:

1. concerning the approach description we provide a detailed description and background information about E2E web testing in general, the problem of parallelizing E2E web test suites in presence of test dependencies and our proposed approach to solve it together with more background information about frameworks and tools required for E2E web test parallelization, such as TestNG and Selenium Grid;
2. concerning the implementation of our tool, we improved STILE by implementing a prioritization mechanism for executing the longest lasting test schedules first in order to reduce the overall execution time. Moreover, we fine-tuned some waiting times that are required for a stable execution of test suites;
3. concerning the empirical study we conducted an evaluation based on real executions of our new parallelization tool, STILE, while in the previous work the effects of parallelization were just estimated using a simplistic mathematical formula. We compared the execution time of STILE not only w.r.t. the sequential execution of a given E2E test suite, but also with the execution time obtained from a reference parallel strategy based on Selenium Grid. We identified the factors affecting the time reduction achievable in practice. Finally, we estimated the reduction in CPU-time w.r.t. the reference parallel strategy;
4. concerning the state of the art we analyzed in detail the existing literature related to the topics of test parallelization, that is the main topic of this work, and test dependency, that is the main problem to solve in order to enable test parallelization.

We provide a Java implementation of our tool, that can be used with Java E2E test suites based on the Selenium WebDriver. The paper is organized as follows. Section 2 illustrates the concepts required for understanding the key notions used in our approach. Section 3 discusses the problem of executing E2E test scripts in parallel. Section 4 describes our approach, while Section 5 presents the empirical study we conducted to evaluate its effectiveness as well as the obtained results. Related works are discussed in Section 6 while Section 7 concludes the paper and gives directions for future work.

## 2. Background

### 2.1. 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 a sequence of actions performed on the web application GUI in order to assess the correctness of a particular feature or functionality. In web testing, test cases are implemented in executable test scripts using testing frameworks (Leotta et al., 2016a) such as Selenium WebDriver (García et al., 2020; Leotta et al., 2025). Selenium WebDriver provides an API for high-level programming languages (like Java and Python) to interact with a web application through a web browser. A test suite is a collection of test scripts which is used to test the various functionalities of the web application under test (WAUT).

Listing 1 shows a (programmable Leotta et al., 2024) test script from an E2E test suite we used as experimental object. Such test script exercises the page that is shown in Fig. 1. 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). 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 (Leotta et al., 2015; Nass et al., 2023) in the GUI. Common examples of locators are IDs, CSS classes 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". Finally, the test scripts can be completed with different types of assertions (Leotta et al., 2020) (not shown in the example).

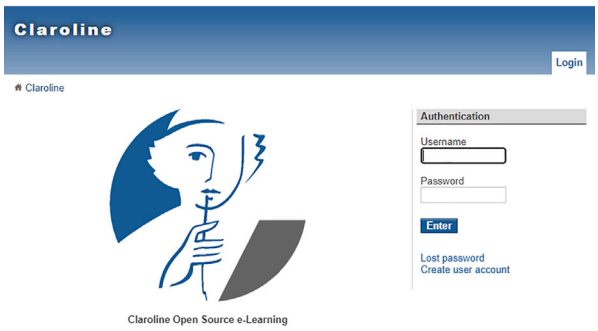


**Fig. 1.** Login page of the Claroline web application.

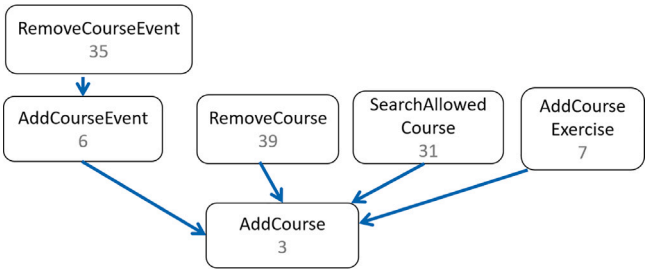


**Fig. 2.** Portion of the test dependency graph generated by TEDD for the test suite of the Claroline Web application.

```
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();
```

Listing 1. E2E test script portion for Claroline Login page (developed in Java).

### 2.2. Dependency among test scripts

A test suite is a set of test cases (ISO/IEC/IEEE, 2022). In the context of order-dependent test suites, Zhang et al. (2014) provide a definition of test suite that enforces an order of execution for test cases: a test suite $T$ is an ordered sequence (i.e., $n$-tuple) of test cases (i.e., $T = \langle t_1, t_2, \ldots, t_n \rangle$) where the index $i$ of each test $t_i$ 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 $t_i$ (Zhang et al., 2014; Gambi et al., 2018; Biagiola et al., 2019). We summarize the relevant definitions in this section in the box below.

**Definition of key testing concepts and their relations**

– *test case*: a sequence of actions performed on a system under test to check if its behavior matches its specifications;
– *test script*: the actual code that implements a test case. In the scope of this work, test case and test script can be used interchangeably;
– *test suite*: in the context of this paper, we assume test suites to be ordered sequences of test cases, (i.e. $T = \langle t_1, t_2, \ldots, t_n \rangle$) where the index $i$ of each test $t_i$ represents the order of execution of each test, as defined by the tester;
– *test dependency*: given a test suite $T = \langle t_1, t_2, \ldots, t_n \rangle$, we say that $t_j$ *depends* on $t_i$ (with $j > i$) if $t_j$ fails when executed before $t_i$;
– *dependency graph*: a graph that represents test dependencies in a test suite, where nodes represent test scripts and edges represent dependencies;
– *warranted schedule*: a warranted schedule is a unique sequence of test scripts that respects all dependencies in the dependency graph, preserving the original order of execution of the test scripts. Given a test script $t_i$, 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;
– *prefix tree*: a prefix tree, also known as trie or digital tree, is a data structure that allows to efficiently store sequences with common prefixes. The common prefixes are stored only once, and the different parts of the sequences are appended as children of the common prefixes. For more information, please refer to Section 4.1.2.;
– *common prefixes of warranted schedules*: two or more warranted schedules share a common prefix if they start with the same test scripts but end with different ones.

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 tests in a test suite should be independent (Muşlu et al., 2011; Zhang et al., 2014; Gambi et al., 2018). Such practice also applies to E2E testing (Ricca and Stocco, 2021), requiring that the result of the execution of a test script should not be affected by other tests executed before it. Having independent tests enables the use of test optimization techniques, such as test suite minimization and parallelization, and it contributes to mitigate the problem of test flakiness (Luo et al., 2014; Parry et al., 2021). 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. Independent tests contribute to preventing flakiness because, as reported by Luo et al. (2014), 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 a 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 test suites that contain dependent tests. Such independence among test scripts is usually achieved using

methods called *test fixtures*. 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 (Umar and Zhanfang, 2019; Gojare et al., 2015), 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 (Biagiola et al., 2019; Zhang et al., 2014). Indeed, executing test fixtures to make each test script independent requires additional operations, as opposed to relying on the state produced by other tests.

### 2.3. Test dependency graph

A *test 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 (Biagiola et al., 2019) has been proposed to automate this process. For this work, we used test dependency graphs computed using TEDD. Fig. 2 shows a portion of the test dependency graph computed by TEDD for one of the test suites used in our evaluation. Such test suite exercises the *Claroline* web application, a web-based learning management system. The complete test suite for Claroline is 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).

### 2.4. Parallel test execution

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 (Bell et al., 2015).

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,[1] 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;
- *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 (Parallelization, 2022).

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.

Selenium Grid (Selenium, 2022) 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.[2]

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 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. If the test suite does have dependencies, the developer must find a way to ensure that during the parallel execution such dependencies are respected. In fact, Selenium Grid manages only the browser, not

[1] TestNG documentation, Parallelism section.

[2] Selenium Grid documentation, Grid architecture section.

the state of the application under test. We will present our solution in Section 3.

Docker is a containerization platform that allows to distribute applications in a portable and efficient way by using containers, which 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[3] offers a functionality called Dynamic Grid, which 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 that in the standard execution mode of Selenium Grid, browsers must be running before test execution starts, and this can possibly consume huge amounts of RAM depending on the number of browsers needed (that is clearly related to the number of tests to execute). On the other hand, we empirically verified that the use of Dynamic Grid might negatively affect the execution time since starting and stopping containers on demand is an overhead that is absent in the standard execution mode. For these reasons, we adopted a solution based on cloning of Docker containers that do not rely on the Dynamic Grid.

## 3. Parallelization of test scripts executions

While there are frameworks, like Selenium Grid, that enable parallel execution of E2E test scripts, some aspects of test parallelization are not managed directly by them. 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 test suite with dependencies in parallel, always requires multiple instances of the WAUT, where an instance of the WAUT is a copy of the application under test, with 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 first describe these two aspects in more detail. Then, we present an architecture for the parallel execution of test scripts that deals with both aspects.

### 3.1. Dealing with test dependencies

For what concerns dependencies between test scripts, we resort to the test dependency graph of the test suite. From the test dependency graph, it is possible to compute the *warranted schedules* of a test suite and execute them in parallel on different machines. 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 Fig. 3, we can see which test scripts (in faded orange) belong to the warranted schedule of 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.

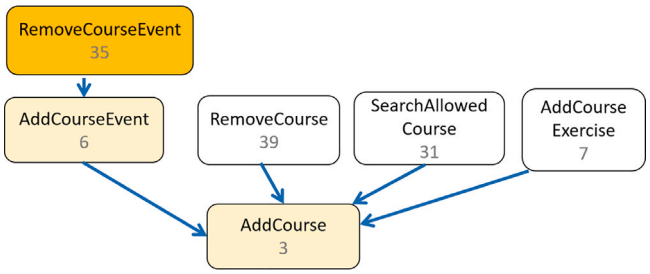


**Fig. 3.** Test scripts composing the warranted schedule (in faded orange) for the `RemoveCourseEvent` test script (in orange). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

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.

### 3.2. Managing the web application state

Another issue that prevents parallel execution of E2E test scripts is the shared state of the web application. In fact, if multiple tests 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 (Kochhar et al., 2019).

### 3.3. Architecture for parallel execution with Selenium Grid

To evaluate the actual benefits that can be achieved by adopting our approach we need a baseline way to execute, in parallel, test suites with dependencies. Such baseline would represent the straightforward solution possibly adopted by practitioners. Thus, we implement parallel execution of warranted schedules using 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.
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,

[3] Selenium Docker.

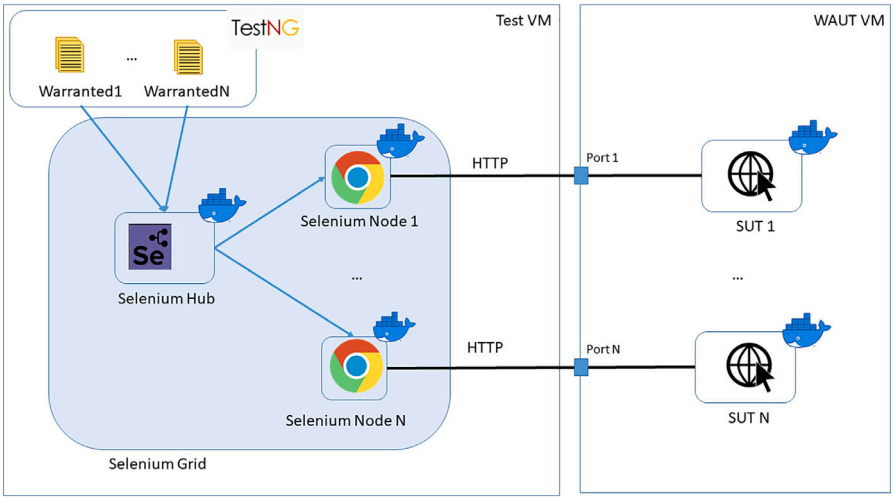


**Fig. 4.** Architecture for parallel execution with Selenium Grid and TestNG.

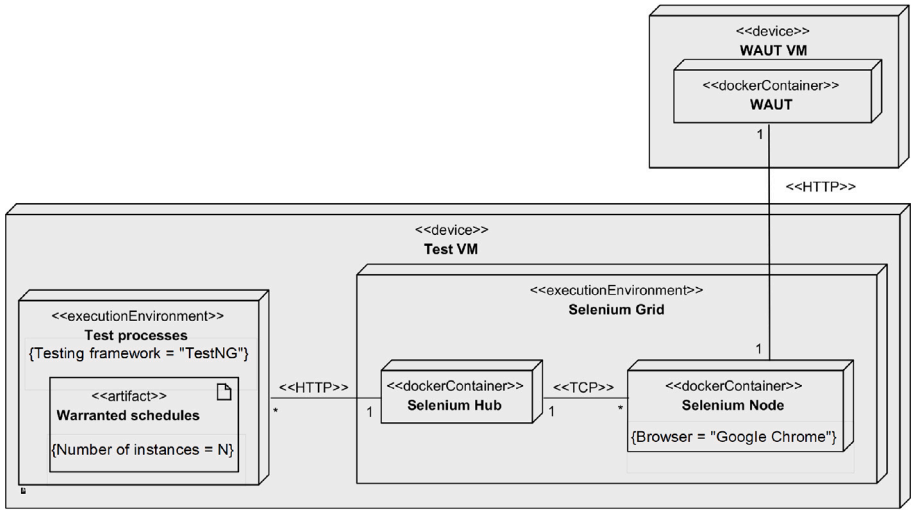


**Fig. 5.** UML deployment diagram for parallel execution with Selenium Grid and TestNG.

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[4] 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. The runtime of the warranted schedules has been computed from past executions. The approach of running the longest task first to reduce the overall execution time is well known in parallel programming (Graham, 1969).

Fig. 4 shows the architecture we use for the parallel execution of test scripts with Selenium Grid, while Fig. 5 represents the same architecture with a UML Deployment Diagram. 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 (Docker, 2022) as it simplifies the deployment, the execution of browsers in Docker containers, and it offers better stability. As shown in Fig. 4, the Test VM also runs the Selenium Grid hub and nodes, in order to reduce costs. 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 number of WAUT instances is equal to the number of warranted schedules. In fact, each warranted schedule must be executed against a dedicated instance of the WAUT.

## 4. Stile: a tool for efficient test scripts parallelization

Depending on the structure of the dependency graph of the WAUT, running all the warranted schedules in parallel is not always 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.

[4] Selenium Grid Docker Images documentation.

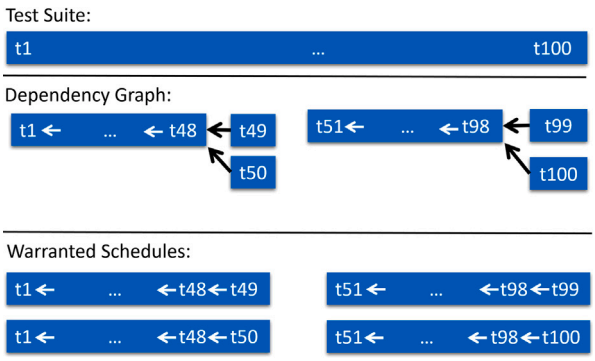


**Fig. 6.** Example of test suite, its associated dependency graph and warranted schedules.

Let us consider an example of a test suite $T$ composed of 100 test scripts (see Fig. 6). 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, we have the four warranted schedules shown at the bottom of Fig. 6. 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.

### 4.1. 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 Fig. 7. The full implementation and replication package of STILE is publicly available at: https://sepl.dibris.unige.it/STILE.php.

#### 4.1.1. Warranted schedules extractor

The first step of our approach (step ❶ in Fig. 7) 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 Fig. 2, 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. While TEDD's dependency graph is an approximation of the ground truth dependency graph, the warranted schedules derived from it respect all tested, manifest dependencies, which ensures their executability with no errors (i.e., these schedules are by construction correct).

#### 4.1.2. Prefix tree builder

The second step of the approach (step ❷ in Fig. 7) 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 De La Briandais (1959), 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 Fig. 8 (extracted from the graph shown in Fig. 2).

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.

#### 4.1.3. Test scripts parallelizer

The third step of the approach (step ❸ in Fig. 7) executes the test scripts of the prefix tree in parallel. To this end, the *Test Scripts Parallelizer* visits the prefix tree by executing Algorithm 1. The algorithm is also represented as a UML Activity Diagram in Fig. 9.

**Algorithm 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          visit(node.children[n-1], container)
8      else if n == 1 then
9          visit(node.child, container)
10     else
11         destroy container
12         return
```

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 Fig. 7 for space reasons. The core component is the *Tree Parallelizer Manager*,

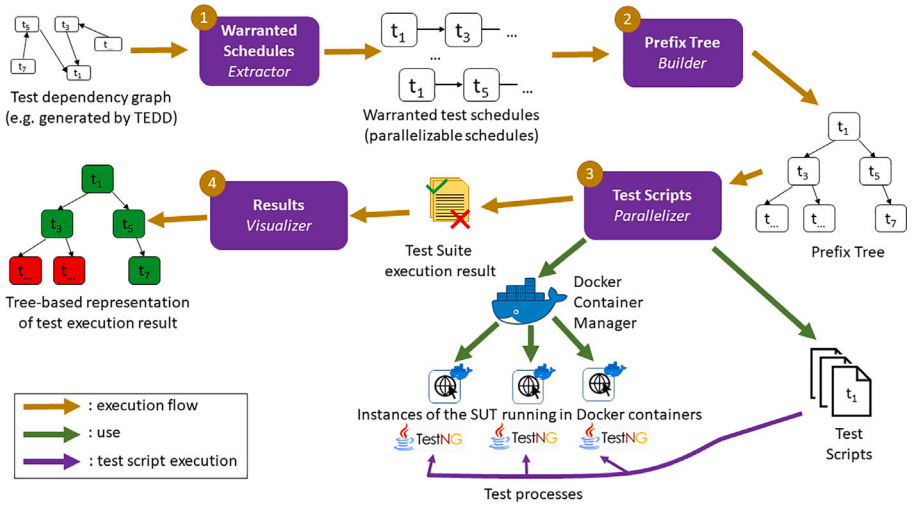


**Fig. 7.** Overview of the four main phases implemented in Stile within four specific modules. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

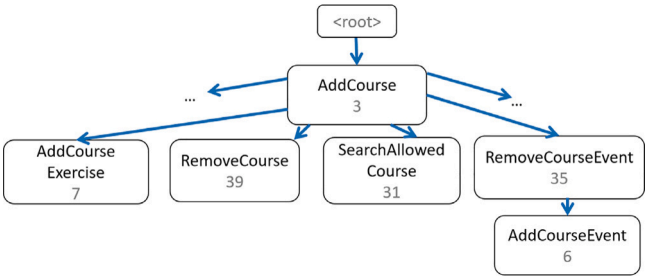


**Fig. 8.** Portion of the Prefix Tree for the Claroline test suite (corresponding to the dependency graph shown in Fig. 2).

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 us see in detail how Algorithm 1 performs the visit. At line 2, it executes the test script associated to the node against its container. Then, it checks how many children the current node has. If it has more than one children, for each child except the last one 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. If the node has only one child (line 10), visit() is called on the only child and on the current container (line 11). 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.

Fig. 10 compares the execution of an example test suite composed of ten test scripts executed either sequentially, with the parallel baseline (i.e., Selenium Grid), or with Stile. In the figure, each cell represents a test script, while the arrows represent the execution order. On the right, we list the dependencies among the test scripts. Those dependencies define the dependency graph, from which we extract the four warranted schedules represented in the Parallel Baseline section of the figure. Finally, the Stile section of the figure represents how the four warranted schedules are executed using Stile. The dummy root node has no test associated, and simply represents the beginning of the execution. The common prefix of the first three warranted schedules ($t_1$, $t_2$, $t_3$) is executed only once, and the main execution is forked twice, to run $t_4$ and $t_6$ in parallel. The last warranted schedule, that does not share a common prefix with the other three schedules, starts at the beginning of the execution, in parallel with the other branch of the tree.

#### 4.1.4. Results visualizer

In the last step (step ❹ in Fig. 7) 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.[5]

### 4.2. 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 Fig. 11, and in Fig. 12 as a UML Deployment Diagram. 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 containers with instances of the WAUT. Those containers are created and destroyed by Stile, that controls the Docker instance running on the WAUT VM.

[5] https://graphviz.org/.

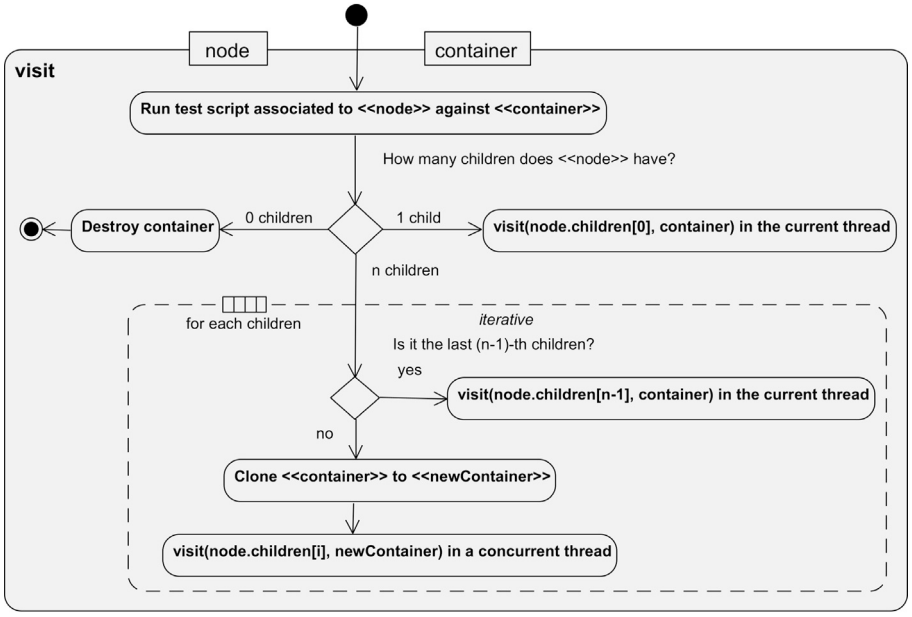


**Fig. 9.** UML activity diagram for the visit algorithm.

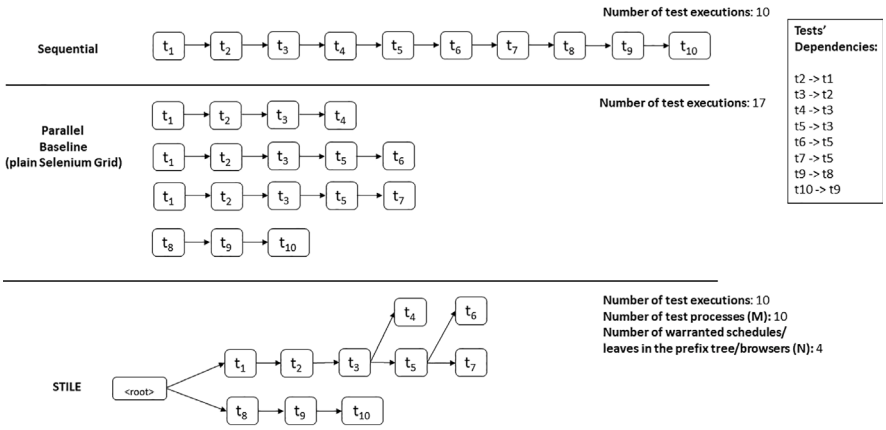


**Fig. 10.** Comparison of the execution of an example test suite composed of seven test scripts and executed Sequentially, with the Parallel Baseline (i.e., Selenium Grid) or STILE.

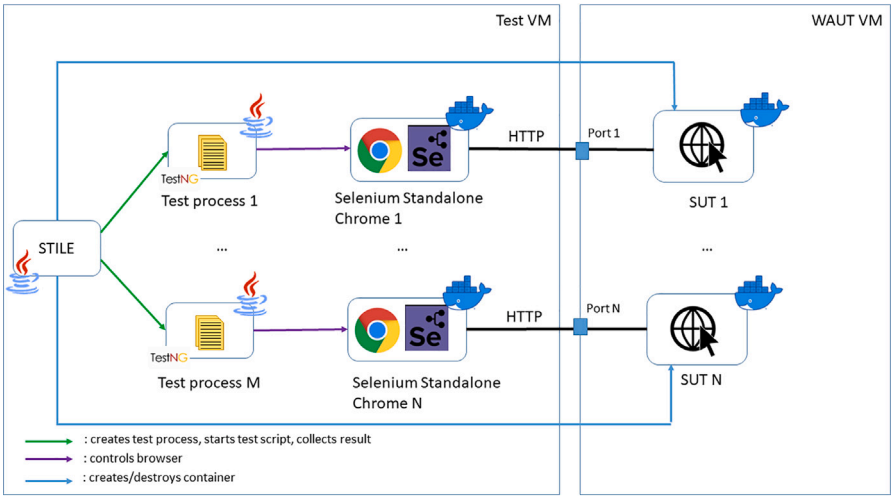


**Fig. 11.** Architecture for the deployment of STILE.

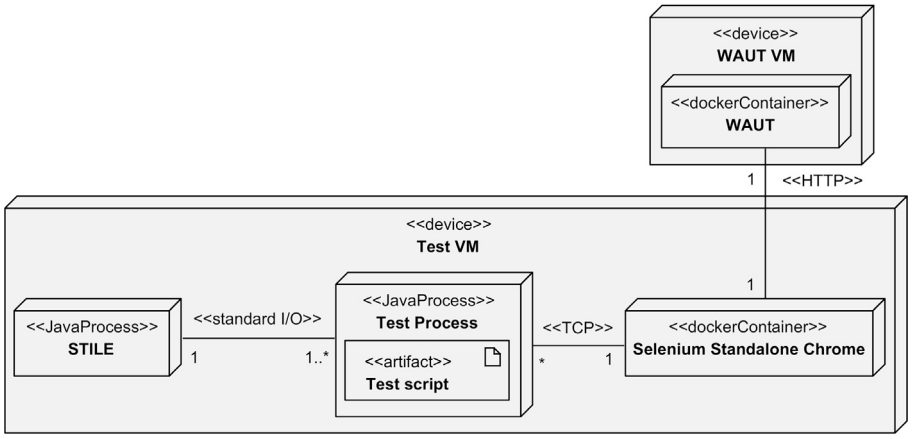


**Fig. 12.** UML deployment diagram for STILE.

Let us consider in more detail the role of the different components shown in Fig. 11. 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. However, it is still possible to configure STILE to run test processes on demand.

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 (Browsers, 2022), although Selenium Standalone docker images are available also for Mozilla Firefox and Microsoft Edge (Docker, 2022).

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 Fig. 11 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 4.1.2. Therefore, we have $M$ test processes, $N$ browsers and $N$ WAUT instances.

The creation of $N$ browsers is an optimization choice to reduce the amount of RAM required to run STILE. In the previous experimental versions of STILE, we used to create $M$ browsers, but this caused a relevant consumption of RAM when we tried to run STILE on large test suites. In fact, even if we cannot statically determine the precise maximum number of test scripts that will be executed simultaneously, we can be sure that this number will never exceed the number of leaves in the tree ($N$, that is also equal to the number of warranted schedules). Therefore, in order to save RAM, we decided to start $N$ browser instances at the beginning of the execution. These browsers are put in a queue and assigned to test processes when required. Similarly to what happens with threads, when the execution of a test script ends, its browser will be passed to the last children of the current graph node, and terminated if the current graph node has no children.

## 5. Empirical study

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. (2012) on designing and reporting empirical studies in software engineering.

### 5.1. 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 3.3.

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, of which six were already used in previous studies (Biagiola et al., 2019; Leotta et al., 2016b, 2021, 2018) and two (Joomla and ExpressCart) have been developed in University classes about software testing. We selected these applications since they: (1) are medium-size applications; (2) are good representatives of typical web applications in terms of functionalities they provide and technologies they use, such as programming languages, databases, libraries and frameworks; and (3) belong to different application domains. We chose to expand the previously used dataset of six applications with Joomla and ExpressCart to further increase the variety of the dataset. In detail, ExpressCart includes different technologies with respect to the classic PHP+MySQL paradigm: it is written in Node.js and uses MongoDB as database. Joomla instead is one of the most used Content Management Systems (CMS) in the world.[6] Table 1 shows relevant information concerning the test suites

[6] https://w3techs.com/technologies/history_overview/content_management/ms/y.

**Table 1**
Experimental objects.

| Application (Test suite) | Test scripts | LOC | Edges in dependency graph | Nodes in prefix tree | Leaves in prefix tree[a] | Longest warranted schedule[b] | Application features coverage |
|---|---|---|---|---|---|---|---|
| Addressbook | 27 | 2502 | 31 | 37 | 17 | 430 | 82% |
| Claroline | 40 | 3109 | 42 | 48 | 28 | 358 | 71% |
| Collabtive | 40 | 2137 | 40 | 44 | 16 | 950 | 77% |
| MantisBT | 64 | 4029 | 58 | 65 | 53 | 188 | 59% |
| MRBS | 22 | 1449 | 21 | 22 | 14 | 456 | 77% |
| PPMA | 23 | 1546 | 22 | 26 | 16 | 291 | 64% |
| Joomla | 85 | 3928 | 66 | 90 | 58 | 343 | 67% |
| ExpressCart | 27 | 2135 | 25 | 27 | 15 | 385 | 69% |
| Average | 41 | 2604,4 | 38,1 | 44,9 | 27,1 | 425,1 | 71% |
| Median | 33,5 | 2319,5 | 35,5 | 40,5 | 16,5 | 371,5 | 70% |
| Min | 22 | 1449 | 21 | 22 | 14 | 188 | 59% |
| Max | 85 | 4029 | 66 | 90 | 58 | 950 | 82% |

Notes:
[a] i.e., width of the prefix tree.
[b] Measured as execution time in seconds (i.e., the depth of the tree).

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), the number of nodes in the prefix tree (column 5) and the coverage of the application features (Walkinshaw et al., 2010) (column 6). To compute the coverage, we listed every feature of the web application under test (e.g., in the Addressbook web application: add a new contact, modify an existing contact, remove a contact, define a group of contacts, etc.) and then checked how many of these features were exercised by our test suites. The coverage never reaches 100%, as most of these applications contain features that require sending and receiving emails or uploading files, which are considerably difficult and time-consuming to test in E2E testing, in particular sending and receiving e-mails which requires asynchronous interaction with third-party services, that since are not in real-time cannot be performed while running a test suite.

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 1 h 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).

### 5.2. 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 4 GB of RAM for each core) we relied on the on-demand cloud computing platform provided by Amazon Web Services[7] (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: we computed the standard deviations of the execution time for all the test suites in all the execution modes (sequential, Grid baseline and Stile) and we found that on average they are negligible (the average standard deviation considering all the test suites and execution modes is 1.08%). Hence, we can reliably estimate the execution time using only three executions.

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

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 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 investigated whether the reduction obtained by increasing the number of cores from 4 to 8 is correlated with the reduction achievable by going from 8 to 32, by using the Pearson correlation coefficient (Pearson, 2008). The rationale behind this analysis is that the existence of a (strong) correlation may justify scaling up the hardware (e.g., to 32 cores) to achieve even higher reduction.

**$RQ_5$ (CPU-time):** *What is the reduction in the CPUs usage (i.e., the overall CPU-time) of Stile when compared with Grid?*

**Table 2**
Time required (in seconds) to execute the E2E test suites with STILE, on different machine configurations, compared to the sequential execution.

| Test suite | seq | | | | STILE | | | |
|---|---|---|---|---|---|---|---|---|
| | 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 | 1478 | 1419 | 1286 | 1255 | 1126 | 899 | 890 | 672 |
| MRBS | 1015 | 944 | 919 | 936 | 748 | 605 | 538 | 510 |
| PPMA | 718 | 709 | 705 | 706 | 492 | 485 | 483 | 483 |
| Joomla | 3448 | 3377 | 3208 | 3112 | 1673 | 790 | 632 | 614 |
| ExpressCart | 1816 | 1779 | 1727 | 1723 | 581 | 459 | 433 | 427 |
| Average | 1529 | 1484 | 1430 | 1413 | 896 | 674 | 617 | 574 |
| Median | 1353 | 1345 | 1290 | 1274 | 833 | 603 | 511 | 496 |
| Min | 718 | 709 | 705 | 706 | 492 | 452 | 433 | 422 |
| Max | 3448 | 3377 | 3208 | 3112 | 1673 | 1102 | 1040 | 1032 |
| SD | 844 | 835 | 787 | 757 | 408 | 235 | 227 | 206 |

The objective of this RQ is to compare the total CPU-time required for the execution of the test suites with STILE and Grid. Under some approximation, CPU-time can be regarded as a proxy for energy consumption. In fact, estimating the energy consumption directly associated to the execution of a test suite is a complex task. Since in the cloud environment in which we ran our experiments, we cannot directly measure the energy consumption of a test suite execution, we can only consider CPU-time. To answer $RQ_5$, we computed the total CPU-time required to run a test suite in STILE or Grid by summing up the execution time of the individual test scripts executed by the two different parallelization strategies, and compared it with the CPU-time required to run the test suite sequentially.

### 5.3. Results

**$RQ_1$ (Execution Time w.r.t. Sequential).** Table 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.

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 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 (492 s vs. 1309 s 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).

From Table 3 we can see that the time saving achievable with STILE ranges from 12% to up to 80% (47%, 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, Joomla STILE reaches a 77% 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).

**Table 3**
Time saving (percentage) achievable with STILE thanks to the parallelization of the E2E test suites w.r.t. the sequential execution.

| 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 | 24% | 37% | 31% | 46% |
| MRBS | 26% | 36% | 41% | 46% |
| PPMA | 31% | 32% | 31% | 32% |
| Joomla | 51% | 77% | 80% | 80% |
| ExpressCart | 68% | 74% | 75% | 75% |
| Average | 37% | 47% | 50% | 53% |
| Median | 29% | 37% | 46% | 50% |
| Min | 12% | 21% | 24% | 25% |
| Max | 68% | 77% | 80% | 80% |
| SD | 21% | 21% | 21% | 20% |

***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. 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 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 80% in the best case (i.e., Joomla). On average, the adoption of STILE allows to halve the execution time of a test suite w.r.t. the sequential execution.

**Table 4**
Time required (in seconds) to execute the E2E test suites with Grid, on different machine configurations, compared to the sequential execution.

| | 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 | 2206 | 1105 | 541 | 400 | 1126 | 899 | 890 | 672 |
| MRBS | 1088 | 715 | 539 | 474 | 748 | 605 | 538 | 510 |
| PPMA | 590 | 438 | 434 | 432 | 492 | 485 | 483 | 483 |
| Joomla | 1668 | 859 | 503 | 416 | 1673 | 790 | 632 | 614 |
| ExpressCart | 1004 | 510 | 451 | 396 | 581 | 459 | 433 | 427 |
| Average | 1380 | 766 | 549 | 486 | 896 | 674 | 617 | 574 |
| Median | 1207 | 732 | 504 | 410 | 833 | 603 | 511 | 496 |
| Min | 590 | 438 | 412 | 396 | 492 | 452 | 433 | 422 |
| Max | 2206 | 1216 | 1006 | 967 | 1673 | 1102 | 1040 | 1032 |
| SD | 585 | 281 | 191 | 196 | 408 | 235 | 227 | 206 |

In conclusion, to answer **RQ$_1$** we can say that the time saving achievable with STILE w.r.t. the sequential execution ranges from 12% to up to 80% (47%, on average). Adopting powerful machines (32 cores in our experiment) allows STILE to reach the highest time savings.

**RQ$_2$ (Time Saving w.r.t. Grid).** Table 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 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.

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 14 out of 16 cases, the only exceptions being Joomla at 4 cores and PPMA at 8 cores), 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 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, computed as $(ET_{Grid} - ET_{STILE})/ET_{Grid}$, where $ET_{Grid}$ is the time for executing the test suite with Grid and $ET_{STILE}$ is the time for executing the test suite with STILE.. 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 50% in Addressbook and an average of 33%. Similarly, STILE is still convenient with an 8 cores machine for most test suites (on average, the time saving is 11%), even though the time savings are less pronounced. When using machines with more than 8 cores, the two approaches become comparable with an 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 8 when answering to RQ5). On the contrary, when the computational resources exceed the number of 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 the cost of cloning containers in STILE becomes significant and negatively affects the execution time. In particular, the overhead is due to the cost of cloning the containers to replicate the state of the system under test, and to the fact that, to improve execution stability and avoid flakiness, all the operations on Docker containers (creation, cloning and termination) are performed in synchronized blocks, and are therefore executed sequentially. However, the difference between STILE and Grid is always very small, with a median of −8% with 32 cores.

**Table 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 | 49% | 19% | −65% | −68% |
| MRBS | 31% | 15% | 0% | −8% |
| PPMA | 17% | −11% | −11% | −12% |
| Joomla | 0% | 8% | −26% | −48% |
| ExpressCart | 42% | 10% | 4% | −8% |
| Average | 33% | 11% | −13% | −20% |
| Median | 37% | 13% | −7% | −8% |
| Min | 0% | −11% | −65% | −68% |
| Max | 50% | 20% | 8% | −5% |
| SD | 18% | 10% | 23% | 24% |

In order to discover which property of the prefix tree has more impact on the execution time reduction obtainable with STILE, we performed a multivariate linear regression analysis using R to understand if there are some properties of the test suites capable of providing a significant effect on the time savings that can be obtained with STILE. Such analysis requires to define two sets of variables (dependent and independent). Based on the intuition that the prefix tree structure associated to each test suite contains valuable information, we decided to include two fundamental properties to define our *independent variables* set (i.e., the set of variables that potentially are able to influence and predict the execution time reduction when varying the number of cores). Intuitively the wider the tree, the more test scripts can be executed simultaneously, so the faster the execution should be. A metric that could approximate the width of the prefix tree is the number of leaves, which in STILE corresponds to the number of different warranted paths (i.e., `n_warranted`). It is related to the maximum parallelization level achievable for the test suite). Moreover, we considered the warranted schedule with the longest runtime (i.e., `longest_warranted`),

relevant to quantify a lower bound for the parallel test suite execution saving and related to the depth of the dependency tree. We also include the number of cores (actually, its logarithm), which also affects the possible saving. The dependent variable $y$ measures the execution time saving. Here are the results of the linear regression:

```
lm(formula = y ~ n_warranted + longest_warranted + log_n_core,
    data = dsperc100)

Residuals:
  Min       1Q   Median       3Q      Max
-32.182  -11.522   -2.889   14.984   28.568

Coefficients:
                  Estimate  Std. Error  t value   Pr(>|t|)
(Intercept)       38.00542    14.47645    2.625     0.0139 *
n_warranted        0.25330     0.20140    1.258     0.2189
longest_warranted -0.03434     0.01599   -2.147     0.0406 *
log_n_core         5.85143     2.72059    2.151     0.0403 *
--
Signif. codes:
0  '***'  0.001  '**'  0.01  '*'  0.05  '.'  0.1  ' '  1

Residual standard error:  17.21 on 28 degrees of freedom
Multiple R-squared:   0.3559, Adjusted R-squared:   0.2869
F-statistic:  5.157 on 3  and  28 DF,  p-value: 0.005781
```

From the table, we can see that the achieved saving $y$ grows when the width of the dependency tree increases (positive coefficient of `n_warranted`), when the depth of the dependency tree decreases (negative coefficient of `longest_warranted`) and as expected when the number of available cores increases (positive coefficient of `log_n_core`). We evaluated the performance as predictor of our regression by performing leave-one-out cross-validation in R, but we obtained a mean absolute error of 15%, too high to make any useful prediction in our specific context. In summary, the schedule with the longest runtime is the only prefix tree property that significantly affects the time saving; however, the resulting linear model is not able to predict the time saving with a reasonable accuracy.

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 (number of cores) and on properties of the test suite (such as width and depth of its prefix tree). We observed savings up to the 50% 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 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 five cases out of eight while in three the savings are slightly lower (i.e., PPMA, Collabtive and MantisBT cases).

***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

**Table 6**
Comparison of the time required in seconds 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 | Grid |
|---|---|---|---|---|---|
| | Complete | Longest schedule | | 32 | 32 |
| | | Time | % | Cores | Cores |
| Addressbook | 1309 | 430 | 67% | 67% | 69% |
| Claroline | 1047 | 358 | 66% | 60% | 61% |
| Collabtive | 1397 | 950 | 32% | 26% | 31% |
| MantisBT | 1478 | 188 | 87% | 55% | 73% |
| MRBS | 1015 | 456 | 55% | 50% | 53% |
| PPMA | 718 | 291 | 59% | 33% | 40% |
| Joomla | 3448 | 343 | 90% | 82% | 88% |
| ExpressCart | 1816 | 385 | 79% | 76% | 78% |
| Average | 1529 | 425 | 67% | 56% | 62% |
| Median | 1353 | 371 | 66% | 57% | 65% |
| Min | 718 | 188 | 32% | 26% | 31% |
| Max | 3448 | 950 | 90% | 82% | 88% |
| SD | 844 | 228 | 19% | 20% | 19% |

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 cases of PPMA and MantisBT the differences are more significant, i.e., for PPMA a difference of 27 and 20 percentage points respectively for Stile and Grid; and for MantisBT a difference of 33 and 14 percentage points respectively for Stile and Grid. One possible explanation is that both the dependency graphs of PPMA and MantisBT have 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 11 and 5 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.

Fig. 13 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

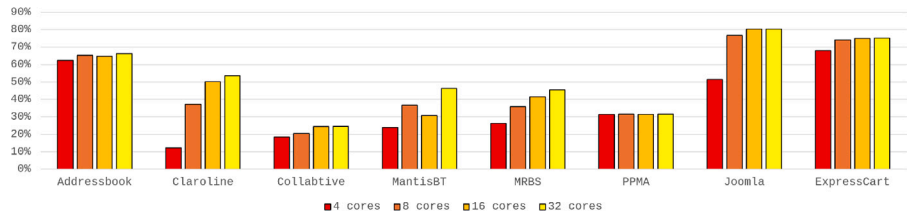


**Fig. 13.** Reduction of test suite execution time when increasing the number of cores (w.r.t. the sequential execution).

**Table 7**
Execution time reductions for the considered test suites, when passing from 4 to 8 cores and from 8 to 32 cores. Pearson correlation coefficient computed between the two columns.

| Test suite | Time reduction | |
|---|---|---|
| | 4 => 8 cores | 8 => 32 cores |
| Addressbook | 3% | 1% |
| Claroline | 30% | 17% |
| Collabtive | 3% | 5% |
| MantisBT | 15% | 15% |
| MRBS | 14% | 9% |
| PPMA | 1% | 0% |
| Joomla | 26% | 5% |
| ExpressCart | 7% | 2% |
| **Pearson correlation coefficient** | **0.731** | |

**Table 8**
Number of test scripts executed with sequential ("seq"), Grid and Stile.

| Test suite | Number (#) of tests executed | | | | | |
|---|---|---|---|---|---|---|
| | seq | Grid | | Stile | | Stile vs. Grid |
| | # tests | # tests | increment | # tests | increment | reduction |
| Addressbook | 27 | 66 | 144% | 37 | 37% | −44% |
| Claroline | 40 | 81 | 103% | 48 | 20% | −41% |
| Collabtive | 40 | 96 | 140% | 44 | 10% | −54% |
| MantisBT | 64 | 196 | 206% | 65 | 2% | −67% |
| MRBS | 22 | 50 | 127% | 22 | 0% | −56% |
| PPMA | 23 | 49 | 113% | 26 | 13% | −47% |
| Joomla | 85 | 124 | 46% | 90 | 6% | −27% |
| ExpressCart | 27 | 45 | 67% | 27 | 0% | −40% |
| Average | 41 | 88 | 118% | 45 | 11% | −47% |
| Median | 34 | 74 | 120% | 41 | 8% | −45% |
| Min | 22 | 45 | 46% | 22 | 0% | −67% |
| Max | 85 | 196 | 206% | 90 | 37% | −27% |
| SD | 23 | 51 | 50% | 23 | 13% | 12% |

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 an 8 core machine, therefore doing this experiment to evaluate the opportunity of buying powerful hardware or cloud services will not have any cost.

Table 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 3. It is evident that there is a 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 Pearson correlation coefficient (Pearson, 2008), and can be observed in Fig. 14 where the regression of the points of Table 7 is presented.

***Discussion***. The results highlight that different test suites show different behaviors when more computational resources are employed. Although we cannot statically predict the execution time reduction when adding more cores, we noted that test suites that benefit from more computational resources (like Claroline, MantisBT, and Joomla) show a high reduction in execution time 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 might be convenient to run the test suite using an even 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$_5$ (CPU-time).** In Table 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 1 lists the names of the test suites associated with the considered application, 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 118%). 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 to the number of tests (Table 8) and to their CPU time (Table 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 two out of eight cases (i.e., 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 six cases, there is a slight increase, ranging from 2% to 37% (on average, the increase in the number of executed test scripts is 11%). Columns 4 and 6, which represent the increment in executed tests with respect to a sequential execution, have been computed as $(Tests_{Grid} - Tests_{seq})/Tests_{seq}$ for Grid and $(Tests_{STILE} - Tests_{seq})/Tests_{seq}$ for Stile.

Column 7 shows the percentage reduction in the number of executed test scripts that can be obtained by using Stile instead of Grid. It has been computed as $(Tests_{STILE} - Tests_{Grid})/Tests_{Grid}$. Such reduction ranges from 27% to 67% (on average, Stile requires to execute -47% test script compared to Grid).

Table 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 41% to 390%, with an average of 170%. On the contrary, the CPU time has a

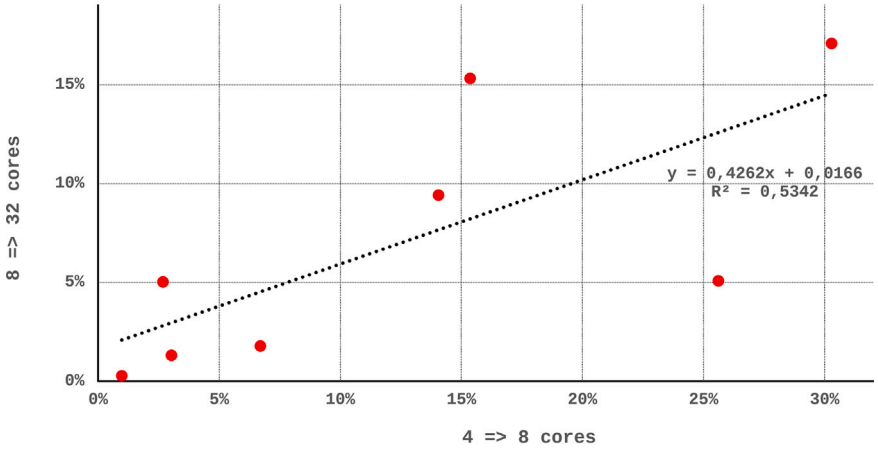


**Fig. 14.** Plot of the reductions from 4 to 8 cores and from 8 to 32 cores as reported in Table 7. Each point represents a test suite. The plotted Linear Regression line is the one that maximizes the $R^2$ fitting value for the eight considered points.

**Table 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 |
| | # tests | # tests | Increment | # tests | increment | reduction |
| Addressbook | 1293 | 3449 | 167% | 1801 | 39% | −48% |
| Claroline | 907 | 2565 | 183% | 1349 | 49% | −47% |
| Collabtive | 1369 | 6703 | 390% | 1681 | 23% | −75% |
| MantisBT | 1255 | 4943 | 294% | 1257 | 0% | −75% |
| MRBS | 936 | 2107 | 125% | 991 | 6% | −53% |
| PPMA | 706 | 1388 | 97% | 779 | 10% | −44% |
| Joomla | 3112 | 4394 | 41% | 3120 | 0% | −29% |
| ExpressCart | 1723 | 2883 | 67% | 1791 | 4% | −38% |
| Average | 1413 | 3554 | 170% | 1596 | 16% | −51% |
| Median | 1274 | 3166 | 146% | 1515 | 8% | −48% |
| Min | 706 | 1388 | 41% | 779 | 0% | −75% |
| Max | 3112 | 6703 | 390% | 3120 | 49% | −29% |
| SD | 757 | 1723 | 118% | 719 | 19% | 16% |

more modest increase when using Stile 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 16% on average. As in the previous table, columns 4 and 6, which represent the increment in CPU-time with respect to a sequential execution, have been computed as $(CPUtime_{Grid} - CPUtime_{seq})/CPUtime_{seq}$ for Grid and $(CPUtime_{STILE} - CPUtime_{seq})/CPUtime_{seq}$ for Stile.

Regarding the comparison between Stile and Grid, Column 6 of Table 9 shows the percentage decrease in CPU time. It has been computed as $(CPUtime_{STILE} - CPUtime_{Grid})/CPUtime_{Grid}$. We can see that, in all case studies, Stile uses less CPU time than Grid, with a reduction ranging from 29% to 75% with an average of 51%. 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 test suite with dependencies in parallel, with respect to the baseline approach using Grid. CPU time can be seen as a proxy for energy consumption (Mahesri and Vardhan, 2005). Indeed, modern CPUs change their clock and voltage accordingly to the workload through Dynamic Frequency and Voltage Scaling (DFVS) (Kim et al., 2008), and this implies that the heavier the workload, the faster the clock and higher the voltages; correspondingly the more energy is consumed. 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. Mahesri and Vardhan (2005) 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 are idle.

> In conclusion, to answer **RQ$_5$** we can say that the reduction in CPU-time of Stile when compared with Grid is significant, ranging from 29% to 75% with an average of 51%.

### 5.4. 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. 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 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 51%), 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 50% 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 CPU-time required to run a test suite, and thus can potentially reduce the 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.

### 5.5. 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 possible flaws in our implementation: although we are confident that major bugs are probably not present in our implementation (i.e., the ones that could lead to catastrophic failures, since we executed many times our tool with many test suites and different settings), 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 chose have already been used in previous studies (Biagiola et al., 2019; Leotta et al., 2016b, 2021, 2018) and belong to different domains. The remaining two, were implemented following the best testing practices for E2E web testing. Hence, we deem them as representatives of general E2E Web test suites. Further experiments on additional subjects would be desirable to corroborate our findings.

*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 (Biagiola et al., 2019) 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).

*Conclusion validity* threats concern issues that affect the ability to draw a correct conclusion. Concerning RQ1, RQ2, RQ3 and RQ5, we simply compared data and percentages.

## 6. Related work

In this section we summarize existing research works related to our study. In particular we analyze the literature on test parallelization, that is the main topic of this work, and test dependency, that is the main problem to solve in order to enable test parallelization.

### 6.1. 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. (2017) 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 (Mondal et al., 2021; Parry et al., 2021). Indeed, running a test suite in parallel may cause non-deterministic behaviors of tests (i.e., flakiness), consisting of tests passing or failing non-deterministically.

In the literature, there are different approaches for parallel execution of test suites with dependencies. For instance, Parsa et al. (2016) 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 (Dorigo et al., 1999) to build near-optimal schedules (from the execution time perspective) that can be executed in parallel. Differently from our work, they do not provide a tool to execute the tests in parallel, but only provide the pseudo-code for the algorithm that computes the scheduling. Their algorithm takes into account aspects that are not considered in our work, like the constraints on available resources. Another difference between their (Parsa et al., 2016) and our approach is that theirs allows parallel executions of different test schedules against the same instance of the application under test (in Figure 1 of the paper (Parsa et al., 2016), the $TE_n$ (Testing Environment) are instances of the application under test, $AG_n$ are schedule executions against the same instance). This is not done in STILE and in the baseline, since our warranted schedules must be executed in isolation to avoid interferences. On the contrary, their schedules (Parsa et al., 2016) are constructed to support such behavior (if two schedules are assigned to the same Testing Environment, it means that there are no constraints that forbid it). 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. (2021). 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. Indeed, their approach does not expect that all the tests pass on the first run, but is instead divided in different steps that requires test re-execution. By relying on warranted schedules derived from TEDD's dependency graph, our approach can instead assume all tests to be successful in the first run, and so it makes sense to us to think that their approach may not be the best solution to parallelize a test suite whose dependencies are known, since in this case an exact approach that expects all the executions to be successful in the first run can be employed. It can be, however, a great solution for the parallelization of test suites whose dependencies are not known. Finally, 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 and Datta (2013) 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. The sets are then prioritized, giving higher priority to the test sets that exercise recently modified functionalities of the application. Differently from our work, 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 when all the tests pass: in fact, to validate their approach they rely on the APFD (Average Percentage of Faults Detected) metric (Rothermel et al., 2001), a metric that measures how fast fault detection occurs in a given test suite, not considering the case where all test scripts pass (i.e., there are no faults).

### 6.2. Test dependencies

The problem of test dependency started to be investigated in the literature only in recent years. In a recent industrial survey (Leotta et al., 2023), our research group discovered that often practitioners highlight a problem in the scalability of E2E test suites: indeed, in their experiences, it is quite difficult to refactor an existing sequential complex test suite to support parallel test script execution, mainly due to the dependencies among test scripts. This means that test suites with dependencies are a real problem at least in complex industrial test environments.

The first empirical analysis of flaky tests by Luo et al. (2014), found that test order dependency is one of the top three common causes of flakiness.

The dependent test detection problem has been formalized by Zhang et al. (2014). They studied 96 dependent tests from five issue tracker systems, formulated the problem and proposed four algorithms to address it, which they implemented in the open source tool DTDetector.

Bell et al. (2015) proposed an approach to detect data dependencies in test suites. In their definition, given an ordered test suite $T = \langle t_1, \ldots, t_n \rangle$ 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.

Note that two tests that are data-dependent may not have a manifest dependency: the definition of manifest dependency given by Zhang et al. (2014) involves test result, while data-dependency regards the access to shared resources. In fact, existing tool and approaches can be divided into two main categories: data-dependency detection tools (Bell et al., 2015) and manifest dependency detection tools (Gambi et al., 2018; Biagiola et al., 2019; Zhang et al., 2014). Techniques of the first category analyze tests looking at accesses to shared resources. Such approaches are sound (i.e., they can find all the data dependencies in a test suite) and efficient but, on the other hand, they may return potentially many false positives (i.e., data-dependencies that are not manifest dependencies and therefore have no effect on the test execution result). Techniques of the second category, instead, focus on tests that actually fail if executed in a certain order. Results of these kind of techniques are accurate (i.e., the dependencies they find are manifest dependencies that actually break the test scripts if not respected), but, on the other hand, such techniques can have false negatives as well as a high execution cost, given that they are based on executing the tests in the test suite in different orders.

Belonging to this last category is the work by Gambi et al. (2018), who proposed Pradet, a tool that combines the precision of DTDetector with the speed of ElectricTest. Similarly to ElectricTest, Pradet collects information about data-dependencies in a given test suite and stores them in a dependency graph, i.e., a directed acyclic graph where nodes represent tests and edges represent the dependencies between them. Data-dependencies stored in the dependency graph are then refined to manifest dependencies through an iterative process. In particular, 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 the graph. Otherwise, it marks the corresponding edge as manifest dependency. This process is repeated until all the data dependencies are removed or become manifest.

Biagiola et al. (2019) presented the first test dependency detection approach for E2E web test suites, implemented in a tool named TEDD. The approach is conceptually similar to Pradet (Gambi et al., 2018) and DTDetector (Zhang et al., 2014), although it has some significant differences to adapt it to the E2E web test suites. Differently from previous approaches that discover data-dependencies by tracking accesses to shared variables in tests (Zhang et al., 2014) or in the system under test (Bell et al., 2015), TEDD infers such information statically from test scripts. Indeed, in the first step, the algorithm retrieves the set of input values $S$ submitted by the test, i.e., values written in the input boxes of the WAUT. Then, the algorithm considers each test case $t_f$ following $t$ and searches whether, 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 test dependency graph. 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. Indeed, we used TEDD as part of Stile in order to detect the dependencies of the given E2E test suites.

More recently, Alakeel (2022) 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.

## 7. Conclusions and future work

In this paper, 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 80% 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 75% less CPU time than the parallel execution based on Grid, enabling a reduction in energy and computation costs as well as environmental impact.

These results show the two main strengths of Stile: in a context where computational resources are limited, Stile enables a reduction of the execution time of the test suites; in a context where resources are unlimited (and thus, the maximum time reduction achievable with parallelization is achieved), Stile reduces the overall CPU-time required (by running the common prefixes of warranted schedules only once), therefore allowing a potential reduction of energy consumption.

In our future work, 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 possible future work is a comparison between Stile and other parallelization techniques, both in terms of execution time and required resources. Moreover we consider interesting also to investigate how to employ Stile on large microservices-based web applications, which would require ways to efficiently manage the state of such a complex architectures. 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.

## CRediT authorship contribution statement

**Dario Olianas:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Maurizio Leotta:** Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Filippo Ricca:** Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Matteo Biagiola:** Writing – review & editing, Conceptualization. **Paolo Tonella:** Writing – review & editing, Conceptualization.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgments

This study was partially carried out within the ‘‘EndGame - Improving End-to-End Testing of Web and Mobile Apps through Gamification’’ project (2022PCCMLF) – Next Generation EU within the PRIN 2022 program (D.D.104 - 02/02/2022 Ministero dell’Università e della Ricerca). This manuscript reflects only the authors’ views and opinions and the Ministry cannot be considered responsible for them.

## Data availability

Data will be made available on request.

## References

Alakeel, A.M., 2022. Dependency detection and repair in web application tests. IAENG Int. J. Comput. Sci. 49 (2).

Bell, J., Kaiser, G., Melski, E., Dattatreya, M., 2015. Efficient dependency detection for safe java test acceleration. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2015, Association for Computing Machinery, New York, NY, USA, pp. 770–781. http://dx.doi.org/10.1145/2786805.2786823.

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

2022. Browsers market share - StatCounter. https://gs.statcounter.com/browser-market-share. (Accessed 18 August 2022).

Candido, J., Melo, L., d’Amorim, M., 2017. 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. In: ASE 2017, IEEE Press, pp. 838–848.

De La Briandais, R., 1959. File searching using variable length keys. In: Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference. In: IRE-AIEE-ACM ’59 (Western), Association for Computing Machinery, New York, NY, USA, pp. 295–298. http://dx.doi.org/10.1145/1457838.1457895.

2022. Docker images for the selenium grid server. https://github.com/SeleniumHQ/docker-selenium. (Accessed 18 August 2022).

Dorigo, M., Di Caro, G., Gambardella, L.M., 1999. Ant algorithms for discrete optimization. Artificial life 5, 137-172. Artif. Life 5, 137–172. http://dx.doi.org/10.1162/106454699568728.

Gambi, A., Bell, J., Zeller, A., 2018. Practical test dependency detection. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation. ICST, IEEE, pp. 1–11.

García, B., Gallego, M., Gortázar, F., Munoz-Organero, M., 2020. A survey of the selenium ecosystem. Electronics 9, 1067.

Garg, D., Datta, A., 2013. Parallel execution of prioritized test cases for regression testing of web applications. In: Proceedings of the Thirty-Sixth Australasian Computer Science Conference. ACSC ’13, vol. 135, Australian Computer Society, Inc., AUS, pp. 61–68.

Gojare, S., Joshi, R., Gaigaware, D., 2015. Analysis and design of selenium WebDriver automation testing framework. Procedia Comput. Sci. (ISSN: 1877-0509) 50, 341–346. http://dx.doi.org/10.1016/j.procs.2015.04.038, https://www.sciencedirect.com/science/article/pii/S1877050915005396. Big Data, Cloud and Computing Challenges.

Graham, R.L., 1969. Bounds on multiprocessing timing anomalies. SIAM J. Appl. Math. 17 (2), 416–429. http://dx.doi.org/10.1137/0117039.

2022. ISO/IEC/IEEE International Standard - Software and Systems Engineering – Software Testing –Part 1:General Concepts. ISO/IEC/IEEE 29119-1:2022(E), pp. 1–60. http://dx.doi.org/10.1109/IEEESTD.2022.9698145.

Kim, W., Gupta, M.S., Wei, G.-Y., Brooks, D., 2008. System level analysis of fast, per-core DVFS using on-chip switching regulators. In: 2008 IEEE 14th International Symposium on High Performance Computer Architecture. IEEE, pp. 123–134.

Kochhar, P.S., Xia, X., Lo, D., 2019. Practitioners’ views on good software testing practices. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice. ICSE-SEIP, IEEE, pp. 61–70.

Leotta, M., Cerioli, M., Olianas, D., Ricca, F., 2020. Two experiments for evaluating the impact of Hamcrest and AssertJ on assertion development. Softw. Qual. J. (SQJ) (ISSN: 0963-9314) 28, 1113–1145. http://dx.doi.org/10.1007/s11219-020-09507-0.

Leotta, M., Clerissi, D., Ricca, F., Tonella, P., 2016a. Approaches and tools for automated end-to-end web testing. In: Memon, A. (Ed.), Adv. Comput. (ISSN: 0065-2458) 101, 193–237. http://dx.doi.org/10.1016/bs.adcom.2015.11.007.

Leotta, M., García, B., Ricca, F., 2025. A family of experiments to quantify the benefits of adopting WebDriverManager and Selenium-Jupiter. Inform. Softw. Technol. (IST) (ISSN: 0950-5849) 178, 107595. http://dx.doi.org/10.1016/j.infsof.2024.107595.

Leotta, M., García, B., Ricca, F., Whitehead, J., 2023. Challenges of end-to-end testing with selenium WebDriver and how to face them: A survey. In: Proceedings of 16th IEEE International Conference on Software Testing, Verification and Validation. ICST 2023, IEEE, pp. 339–350. http://dx.doi.org/10.1109/ICST57152.2023.00039.

Leotta, M., Ricca, F., Marchetto, A., Olianas., D., 2024. An empirical study to compare three web test automation approaches: NLP-based, programmable, and capture & replay. J. Softw. Evol. Process (JSEP) (ISSN: 2047-7481) 36 (5), e2606. http://dx.doi.org/10.1002/smr.2606.

Leotta, M., Ricca, F., Tonella, P., 2021. SIDEREAL: Statistical adaptive generation of robust locators for End-to-End Web testing. J. Softw. Test. Verif. Reliab. (STVR) (ISSN: 1099-1689) http://dx.doi.org/10.1002/stvr.1767.

Leotta, M., Stocco, A., Ricca, F., Tonella, P., 2015. Using multi-locators to increase the robustness of web test cases. In: Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation. ICST 2015, IEEE, ISBN: 978-1-4799-7125-1, pp. 1–10. http://dx.doi.org/10.1109/ICST.2015.7102611.

Leotta, M., Stocco, A., Ricca, F., Tonella, P., 2016b. ROBULA+: An algorithm for generating robust XPath locators for web testing. J. Softw. Evol. Process (JSEP) (ISSN: 2047-7481) 28 (3), 177–204. http://dx.doi.org/10.1002/smr.1771.

Leotta, M., Stocco, A., Ricca, F., Tonella, P., 2018. PESTO: Automated migration of DOM-based web tests towards the visual approach. In: Offutt, J., Hierons, R.M. (Eds.), J. Softw. Test. Verif. Reliab. (STVR) (ISSN: 1099-1689) 28 (4), e1665. http://dx.doi.org/10.1002/stvr.1665.

Luo, Q., Hariri, F., Eloussi, L., Marinov, D., 2014. An empirical analysis of flaky tests. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: FSE 2014, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450330565, pp. 643–653. http://dx.doi.org/10.1145/2635868.2635920.

Mahesri, A., Vardhan, V., 2005. Power consumption breakdown on a modern laptop. In: Falsafi, B., VijayKumar, T.N. (Eds.), Power-Aware Computer Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 165–180.

Mondal, S., Silva, D., d'Amorim, M., 2021. Soundy automated parallelization of test execution. In: 2021 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 309–319. http://dx.doi.org/10.1109/ICSME52107.2021.00034.

Muşlu, K., Soran, B., Wuttke, J., 2011. Finding bugs by isolating unit tests. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. In: ESEC/FSE '11, Association for Computing Machinery, New York, NY, USA, pp. 496–499. http://dx.doi.org/10.1145/2025113.2025202.

Nass, M., Alégroth, E., Feldt, R., Leotta, M., Ricca, F., 2023. Similarity-based web element localization for robust test automation. ACM Trans. Softw. Eng. Methodol. (TOSEM) (ISSN: 1049-331X) 32 (3), http://dx.doi.org/10.1145/3571855.

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

2022. Parallelization with TestNG. https://santiautomation.github.io/miscellaneous/2020/09/13/TestNG-Parallelization.html. (Accessed 17 August 2022).

Parry, O., Kapfhammer, G.M., Hilton, M., McMinn, P., 2021. A survey of flaky tests. ACM Trans. Softw. Eng. Methodol. (TOSEM) 31 (1), 1–74.

Parsa, M., Ashraf, A., Truscan, D., Porres, I., 2016. On optimization of test parallelization with constraints. In: Zimmermann, W., Alperowitz, L., Brügge, B., Fahsel, J., Herrmann, A., Hoffmann, A., Krall, A., Landes, D., Lichter, H., Riehle, D., Schaefer, I., Scheuermann, C., Schlaefer, A., Schupp, S., Seitz, A., Steffens, A., Stollenwerk, A., Weißbach, R. (Eds.), Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Wien, 23.-26. Februar 2016. In: CEUR Workshop Proceedings, vol. 1559, CEUR-WS.org, pp. 164–171, http://ceur-ws.org/Vol-1559/paper21.pdf.

Kirch, W. (Ed.), 2008. Pearson's correlation coefficient. In: Encyclopedia of Public Health. Springer Netherlands, Dordrecht, pp. 1090–1091. http://dx.doi.org/10.1007/978-1-4020-5614-7_2569.

Radziwill, N., Freeman, G., 2020. Reframing the test pyramid for digitally transformed organizations. ArXiv.

Ricca, F., Stocco, A., 2021. 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.

Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J., 2001. Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng. 27 (10), 929–948.

2022. Selenium grid official website and documentation. https://www.selenium.dev/documentation/grid/. (Accessed 17 August 2022).

Umar, M.A., Zhanfang, C., 2019. A study of automated software testing: Automation tools and frameworks. Int. J. Comput. Sci. Eng. (IJCSE) 6, 217–225.

Walkinshaw, N., Bogdanov, K., Derrick, J., Paris, J., 2010. Increasing functional coverage by inductive testing: A case study. In: Petrenko, A., Simão, A., Maldonado, J.C. (Eds.), Testing Software and Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 126–141.

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

Zhang, S., Jalali, D., Wuttke, J., Muslu, K., Lam, W., Ernst, M.D., Notkin, D., 2014. Empirically revisiting the test independence assumption. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. In: ISSTA 2014, ACM, New York, NY, USA, pp. 385–396. http://dx.doi.org/10.1145/2610384.2610404, http://doi.acm.org/10.1145/2610384.2610404.

**Dario Olianas** is Postdoc at the University of Genova, Italy. He received his Ph.D. in Computer Science and Systems Engineering in 2023 with a thesis parallelization of dependent test suites and flakiness prevention. He is author or coauthor of some papers published in international journals and conferences/ workshops.

**Maurizio Leotta** is Assistant Professor at the University of Genova, Italy. He received his Ph.D. degree in Computer Science from the same university, in 2015, with the thesis "Automated Web Testing: Analysis and Maintenance Effort Reduction". He is author or coauthor of more than 110 research papers published in international journals and conferences/workshops. His current research interests are in software engineering, with a particular focus on the following themes: Web, Mobile, and IoT application testing, functional test automation, empirical software engineering, business process modeling and model-driven software engineering.

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

**Matteo Biagiola** is Postdoc at the Software Institute of Università della Svizzera italiana (USI) in Lugano, Switzerland. He obtained his Ph.D. in 2020 from Università degli Studi di Genova, Italy, in a joint collaboration with Fondazione Bruno Kessler, Trento, Italy. He is interested in software testing, with a particular focus on test generation for Web applications and learning-based systems. He serves as a reviewer for major Software Engineering conferences and journals, and was awarded several distinguished reviewer awards (i.e., ICSME 2023, ICST 2024, TOSEM 2023).

**Paolo Tonella** is Full Professor at the Faculty of Informatics and at the Software Institute of Università della Svizzera italiana (USI) in Lugano, Switzerland. He is Honorary Professor at University College London, UK. Paolo Tonella holds an ERC Advanced grant as Principal Investigator of the project PRECRIME. He has written over 150 peer reviewed conference papers and over 50 journal papers. In 2011 he was awarded the ICSE 2001 MIP (Most Influential Paper) award, for his paper: "Analysis and Testing of Web Applications". His Hindex (according to Google scholar) is 68. He is/was in the editorial board of TOSEM, TSE and EMSE. He was Program Co-Chair of ESEC/FSE 2023, and he is Program Co-Chair of ISSTA 2025. His current research interests are in software testing, in particular approaches to ensure the dependability of machine learning based systems, automated testing of cyber physical systems, and test oracle inference and improvement.