Dipartimento di Informatica, Bioingegneria,
Robotica ed Ingegneria dei Sistemi

Test Generation and Dependency Analysis for Web Applications

by

Matteo Biagiola

Theses Series **DIBRIS-TH-2019-XX**

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

Università degli Studi di Genova

Dipartimento di Informatica, Bioingegneria,

Robotica ed Ingegneria dei Sistemi

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

Test Generation and Dependency Analysis for Web Applications

by

Matteo Biagiola

December, 2019

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

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

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

Submitted by Matteo Biagiola
DIBRIS, Univ. di Genova
biagiola@fbk.eu

Date of submission: October 2019

Title: Test Generation and Dependency Analysis for Web Applications

Advisors:
Paolo Tonella
Full Professor
Faculty of Informatics and Software Institute
Università della Svizzera Italiana
paolo.tonella@usi.ch

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

Ext. Reviewers:
Ali Mesbah
Associate Professor
Department of Electrical and Computer Engineering
University of British Columbia
amesbah@ece.ubc.ca

Leonardo Mariani
Full Professor
Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli studi di Milano Bicocca
leonardo.mariani@unimib.it

Abstract

In web application testing existing model based web test generators derive test paths from a navigation model of the web application, completed with either manually or randomly generated inputs. Test paths extraction and input generation are handled separately, ignoring the fact that generating inputs for test paths is difficult or even impossible if such paths are infeasible.

In this thesis, we propose three directions to mitigate the path infeasibility problem. The first direction uses a search based approach defining novel set of genetic operators that support the joint generation of test inputs and feasible test paths. Results show that such search based approach can achieve higher level of model coverage than existing approaches.

Secondly, we propose a novel web test generation algorithm that pre-selects the most promising candidate test cases based on their diversity from previously generated tests. Results of our empirical evaluation show that promoting diversity is beneficial not only to a thorough exploration of the web application behaviours, but also to the feasibility of automatically generated test cases. Moreover, the diversity based approach achieves higher coverage of the navigation model significantly faster than crawling based and search based approaches.

The third approach we propose uses a web crawler as a test generator. As such, the generated tests are concrete, hence their navigations among the web application states are feasible by construction. However, the crawling trace cannot be easily turned into a minimal test suite that achieves the same coverage due to test dependencies. Indeed, test dependencies are undesirable in the context of regression testing, preventing the adoption of testing optimization techniques that assume tests to be independent. In this thesis, we propose the first approach to detect test dependencies in a given web test suite by leveraging the information available both in the web test code and on the client side of the web application. Results of our empirical validation show that our approach can effectively and efficiently detect test dependencies and it enables dependency aware formulations of test parallelization and test minimization.

Table of Contents

Chapter 1

Introduction

Web applications are one of the fastest growing class of software systems in use today. They cover a wide range of domains such as banking, e-commerce, healthcare management and, as such, they have a big impact on all aspects of our society. According to a recent survey [1] there are 1.5 billion websites on the world wide web. The number of users worldwide is around 4.3 billion [2], which is roughly 57% of the world population. Thus, web application failures potentially impact many users and the consequences of such failures spread from obvious costs such as revenue lost due to customers being unable to use the product to indirect financial costs coming from problems with brand reputation and customer loyalty when a software failure is made public. A reasonable level of confidence in the software can be gained by testing it before its release. Software testing is nowadays a widely adopted practice in the software industry. Companies like Facebook and Google invest resources in software testing and develop their own tools to automate the software testing process [3].

Modern web applications available nowadays provide the same high level of user interaction as native desktop applications, while eliminating the need for in-site deployment, installation, and update. For instance, single page web applications (SPA) achieve high responsiveness and user friendliness by dynamically updating the Document Object Model (DOM) of a single web page by means of JavaScript functions that react asynchronously to user events. To test such complex software systems, engineers typically adopt test automation frameworks such as *Selenium WebDriver* [4]. In this context, the tester verifies the correct functioning of the web application under test using test cases that automate the set of manual operations that the end user would perform on the web application's graphical user interface (GUI), such as delivering events with clicks, or filling in forms [Bin96, FG99, SYM18, LSRT16, LSRT15, LSRT18, HRS16] (such GUI tests are

[1] `https://www.internetlivestats.com/total-number-of-websites/`
[2] `https://www.internetworldstats.com/stats.htm`
[3] `https://screenster.io/software-testing-facebook-google/`
[4] `http://www.seleniumhq.org`

often called end to end (E2E) tests). Testers implement business-focused test scenarios within such test cases, along with the necessary input data. Each test case, hence, exercises a specific test path along the model of the web application which specifies its functionalities (often called navigation model or navigation graph since a test case *navigates* among the functionalities of the web application under test specified in the model).

1.1 Motivations

Software testing accounts for a significant portion of the overall development cost of a piece of software mainly due to the effort spent by test engineers in generating test data that exercise the system in various ways [PY08]. For this reason, automating test case generation has been the subject of a large body of research work resulting in several techniques and tools.

In the web domain, model based approaches in web testing have received a significant amount of attention from the research community [RT01, MvDR12, MFM13, MTR08, TMN+12, MTR12, TNM+13]. Indeed, models represent the behaviours of the web application abstracting from the underneath technologies and programming languages that shape the web application. Test scenarios represent navigation among the states of the web application under test and are derived by such models using extraction algorithms that try to satisfy some coverage criterion (e.g. making sure that all the transitions of the model are exercised at least once). However, testing research is predominately focused on how to automatically build the model of the web application under test without considering that extracting test paths from the model and generating proper input values for such paths can be difficult or even impossible if the paths are *infeasible*. Among the different open problems in the attempt of automating the web testing process, the *feasibility problem* is a major one. Indeed, infeasible tests violate dependencies and constraints of the web application under test and, as a consequence, do not exercise the functionalities of the web application they are created for. Therefore, solving the feasibility problem would guarantee an adequate coverage of the functionalities of the web application under test, hence a more effective testing process.

Determining if a test path is feasible (a priori, i.e. before execution) is in general undecidable and in practice a very challenging problem. In this thesis, we tackled the feasibility problem under three different point of views, summarized in Figure 1.1. The first direction we considered is to apply search based approaches to look for feasible paths in the model of the web application under test. Secondly, we considered the diversity based approach with the aim of exploring the space of the possible paths at large to find the feasible ones. The third perspective we considered is to use a crawler to directly generate test cases without using a model of the web application under test, such that generated tests are not affected by the feasibility problem. Indeed, web crawlers explore the web application under test by accumulating different navigations that represent different test scenarios. However, such test scenarios are *dependent*. In fact a test case generated during crawling cannot be successfully executed in isolation because it potentially

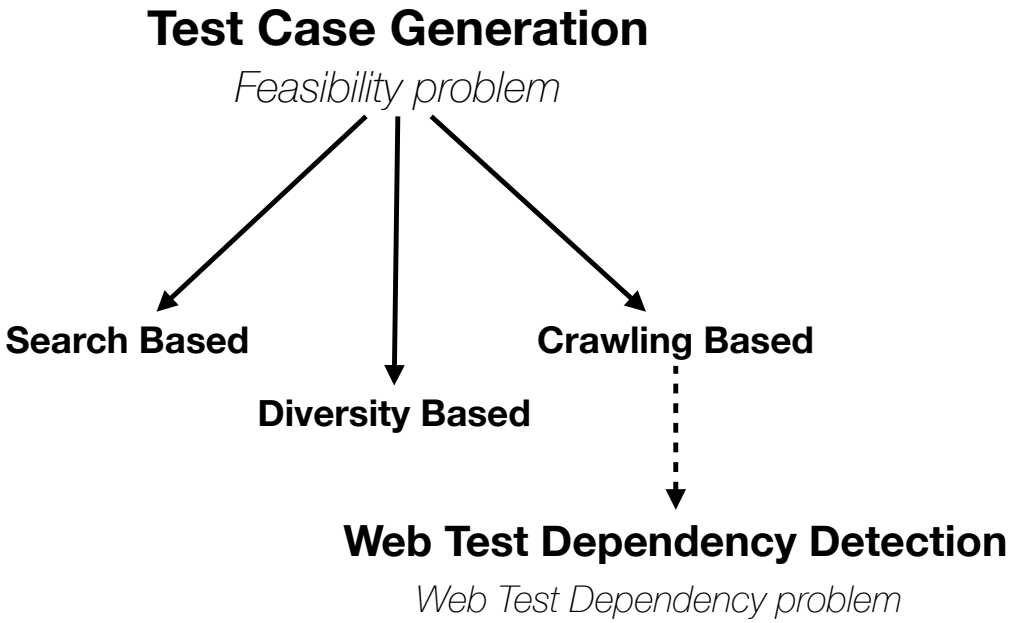


Figure 1.1: Research directions

needs the web application to be in a certain state, produced by the previously executed tests. Despite test dependencies can be useful to exercise new behaviours of the web application under test during test generation, they are not desirable in the context of regression testing since they inhibit the application of test optimization techniques (e.g. test parallelization) that assume tests to be independent.

Therefore, in order to have a functional test suite to be executed during regression testing, crawling based test generators need test dependencies to be resolved. However, the problem of test dependency detection in the web domain is not yet solved by existing approaches [ZJW+14, BKMD15, GBZ18]. Indeed such approaches extract dependencies by detecting, statically or dynamically, read-after-write operations on the shared state between the tests, which is usually represented by static object fields in Java classes. Such analysis cannot be easily performed in E2E web test suites where the state is spread among multiple layers that involve multiple languages and technologies.

1.2 Problem Statement

The feasibility problem deals with automatically finding proper paths in the navigation model of the web application under test along with associated input values, such that, upon execution of the web application under test, the desired navigation is taken.

Infeasibility in model based testing comes from abstracting away the implementation details of web application under test with the aim of getting a concise and useful representation of the web application behaviours in a model. In fact, the abstraction also strips away part of the program semantics and, as a result, tests extracted from the model violate dependencies and constraints that hold in the real web application under test, but they are not represented explicitly in the model.

Let us take the example of an e-commerce web application which has two states, namely the product list state and the shopping cart state. The product list has two possible actions, namely the possibility to go to the shopping cart and the action to add a product to the cart. Moreover, the shopping cart state has only one action, which is checkout. The constraint that is not represented in the model is the fact that the checkout action is possible only if there is at least one product in the cart (it can be noticed that such constraint is *dynamic*, i.e. it is possible to verify its validity only at runtime since the dependencies between the actions are also not represented in the navigation model). Under such assumptions the path in the model composed of the two actions go to cart and checkout is infeasible since it is not possible to buy a product that is not added to the cart. However, the path that includes the action add to cart as first action is a feasible one because the preconditions of the action checkout in the shopping cart state are satisfied.

As the example above shows, practically, infeasibility regards the impossibility to generate input values satisfying the path constraints in the test paths extracted from the model. Therefore, infeasibility hinders an adequate coverage of the web application functionalities since infeasible test paths do not exercise the functionalities they are extracted for (infeasible test paths cannot be executed against the web application under test). The combination of test path extraction and input generation strategies determines the feasibility of a test path, although existing approaches treat them separately, assuming that each extracted test path is feasible. Moreover, random input strategies may be ineffective when decoupled from test paths extraction and manual input generation is costly and limited by the testing resources (e.g. time) available to testers. Therefore, we need novel approaches to address the *feasibility problem* which are able to jointly generate test paths and the corresponding input values, discarding infeasible test paths and evaluating as few test candidates as possible, since E2E tests are characterized by slow execution times due to the interaction with the browser.

The feasibility problem can also be addressed by directly generating concrete test cases without passing through the model based testing process, i.e. extracting abstract test cases (i.e. test paths) from a model and making them executable by generating proper input values. An

appealing approach to generate concrete tests is the crawling based approach where the web crawler produces test cases while exploring the web application under test. However, deriving a functional test suite from the crawling trace is not trivial. Existing approaches [MvD09] *segment* the long crawling trace produced during the exploration, which can be seen as a single test case, into meaningful test cases with the purpose of increasing readability and maintainability. Such segmented test cases are supposed to be executed one after the other since they come from the crawling trace, which is a continuous sequence of events. Therefore, each test case potentially *depends* on the state produced by those executed before it. If such dependencies are not known, during regression testing the generated test suite has to be executed in the order in which the tests were generated, preventing the application of optimization techniques such as test parallelization [BK14], test prioritization [RUCH01], test selection [GEM15] and test minimization [VSM18], which all require having independent test cases. Hence, the crawling based approach for test case generation guarantees feasibility but it is affected by the *web test dependency problem*:

> *The web test dependency problem concerns resolving the dependencies among tests such that the execution of the given test suite can be optimized.*

Test dependency can be informally defined as follows. Let $T = \langle t_1, t_2, \ldots, t_n \rangle$ be a test suite, where each t_i is a test case, whose index i defines an order relation between test cases that corresponds to the original execution order given by testers. When tests within T are executed in the original order, all tests execute correctly. If the original execution ordering is altered, e.g., by executing t_2 before t_1, and the execution of t_2 fails, we can say that t_2 *depends on* t_1 for its execution, and that a *manifest test dependency* exists [ZJW+14, GBZ18].

Ideally, all tests, both automatically generated and manually created, in a test suite should be independent or dependencies between tests should be known and documented. Besides inhibiting the application of test optimization techniques, test dependencies have other implications as well. In fact, test dependencies can mask program faults and lead to undesirable misleading side-effects, such as tests that pass when they should fail, tests that fail when they should pass [ZJW+14, LZE15], or significant test execution overheads [BKMD15, Kap16].

Automated detection of all test dependencies in any given test suite is challenging and it has been proven to be NP-complete [ZJW+14]. As such, researchers have proposed techniques and heuristics that help developers detect an approximation of such dependencies in a timely manner [GSHM15, ZJW+14, GBZ18, BKMD15, Kap16]. However, existing approaches [ZJW+14, BKMD15, GBZ18] are not applicable to E2E web test suites because they are based on the extraction of read/write operations affecting shared data (e.g., static fields) of Java objects. Instead, web applications are prone to dependencies due to the persistent data managed on the server-side and the implicit shared data structure on the client-side represented by the DOM. Such dependencies are spread across multiple tiers/services of the web application architecture and are highly

dynamic in nature. Therefore, the *web test dependency problem* demands for novel approaches that leverage the information available both in the web test code and on the client side of the web application under test to automatically detect test dependencies. Once test dependencies are known, test optimization techniques can be reformulated in order to be applied to dependent test suites.

1.3 Objectives

The objectives of this thesis are as follows:

- Devise new model based test case generation methods for E2E web application testing in order to address the feasibility problem;
- Empirically investigate the performance (i.e. effectiveness and efficiency in terms of coverage and fault detection) of the devised test case generation techniques with respect to state-of-the-art crawling based approaches;
- Devise new test dependency detection techniques for E2E web test suites in order to address the test dependency problem;
- Empirically investigate the performance (i.e. efficiency in terms of time spent to find the dependencies) of the devised test dependency detection techniques;
- Turn a web crawler into a test case generator in order to overcome the feasibility problem which affects the model based test case generators. To that aim, devise new methods to solve the test minimization problem taking into account test dependencies;
- Empirically investigate the performance (i.e. effectiveness in terms of coverage) of the crawling based dependency aware test generator with respect to state-of-the-art crawling based test generators.

1.4 Organization of the Thesis

The remainder of the thesis is organized as follows:

Chapter 2. This chapter presents the concepts related to web application testing that we used in the thesis. It starts with introducing web applications, concepts related to software testing in general and metaheuristic algorithms. Then, it introduces the web application testing domain with a particular focus on E2E testing. Furthermore, the test dependency problem is presented.

Finally, it introduces automation in the context of web application testing with a focus on model based testing, it explains the functioning of a web crawler and how it is used in state-of-the-art approaches to test the functionalities of web applications.

Chapter 3. This chapter provides an overview of the state of the art in the area of web testing. We first present works related to test case generation by describing both approaches for Java software and for web applications. Then, we introduce how the test dependency problem has been addressed by previous approaches with a focus on dependency detection techniques and test optimization techniques that assume test independence.

Chapter 4. This chapter describes our model based approach to test case generation for web applications. The chapter starts with our overall approach and it explains the differences with the crawling based approaches. Then, each component of the approach is described, starting from the testing model extraction. Afterwards, the test generation problem is defined together with the test path feasibility problem. We address such problem in two ways, namely by using search based and diversity based methods. Specifically, the search based web test generation approach is explained and evaluated against the state-of-the-art crawling based approach in terms of size of the generated test suite and transition coverage of the navigation graph. Then, we identify the limitations of search based approaches and we explain how such limitations can be addressed by adopting a diversity based approach. Finally, the diversity based web test generation method is presented and extensively evaluated with respect to the state-of-the-art crawling based approach and the previously presented search based method in terms of effectiveness and efficiency (transition coverage, client side code coverage and fault detection).

Chapter 5. This chapter outlines our test dependency detection approach for web applications. In this chapter we focus on manually written E2E web test suites. First of all, we introduce a running example to help understanding the test dependency problem and the steps of our approach which are presented next. Each step of the approach is detailed, namely dependency graph extraction, filtering, validation and recovery and disconnected dependency recovery. Different techniques are proposed for dependency graph extraction and filtering and all of them are evaluated empirically to establish which combination performs better. The baseline for comparison is the state-of-the-art approach for dependency detection proposed for Java software. The comparison between the combinations of our approach and the baseline is done in terms of time to complete the detection of the dependencies. Furthermore, we show that dependency detection enables test parallelization with a relevant test suite runtime speed-up.

Chapter 6. This chapter reports our dependency aware test case generation approach which uses a web crawler. The test dependency and test redundancy problems are presented with an example. We then outline the steps of our approach, namely test dependency detection and SAT solver based test minimization. Since a web crawler is used to generate tests, we adapt the dependency detection techniques presented in the previous chapter and apply them to automatically generated test cases. Once the dependencies among the generated tests are found we can formulate the dependency aware test minimization problem. We use a SAT solver to minimize the generated

test suite, which ensures that all redundant test cases are removed and all dependencies among tests are respected. Finally, we present the evaluation of such approach with respect to the state-of-the-art crawling based test generator in terms of client side code coverage.

Chapter 7. This chapter concludes the thesis by summarizing its contributions and outlining possible directions for future work.

1.5 Origin of Chapters

Some of the work presented in this thesis has been previously published in the following papers:

- Matteo Biagiola, Filippo Ricca, Paolo Tonella "Search based path and input data generation for web application testing", in *International Symposium on Search Based Software Engineering* (SSBSE 2017), pages 18–32, 2017. This paper contains part of Chapter 4.
- Matteo Biagiola, Andrea Stocco, Filippo Ricca, Paolo Tonella "Diversity-based Web Test Generation", in *Proceedings of 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESEC/FSE 2019), pages 142–153, 2019. This paper contains part of Chapter 4.
- Matteo Biagiola, Andrea Stocco, Filippo Ricca, Paolo Tonella "Web test dependency detection", in *Proceedings of 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESEC/FSE 2019), pages 154–164, 2019. This paper contains part of Chapter 5.

Chapter 6 of this thesis has been accepted and it will be published next year as the following paper:

- Matteo Biagiola, Andrea Stocco, Filippo Ricca, Paolo Tonella "Dependency-Aware Web Test Generation", in *Proceedings of 13th IEEE International Conference on Software Testing, Verification and Validation* (ICST 2020), 2020.

Chapter 2

Web Application Testing: Background

This chapter presents the basic concepts that are extensively used in this thesis. Section 2.1 briefly describes the functioning of web applications and the topic of software testing in general. Furthermore, metaheuristic algorithms are also introduced. Section 2.2 contextualizes software testing for web applications, presenting the different testing approaches with a particular focus on *end-to-end* (E2E) testing and one of the best practices used to develop such test cases, namely the *Page Object Design Pattern*. Moreover, it also defines the test dependency problem in general and in the context of web applications. Section 2.3 explains what it means for web applications to be tested automatically given a *model* of the web application under test (WAUT). Additionally, this section describes how a *web crawler* works and how it is used for testing the functionalities of a web application. The reader who is already familiar with such concepts can safely skip this chapter.

2.1 Preliminaries

2.1.1 Web Applications

The world wide web has changed a lot since its invention in 1989 by Tim Berners-Lee, shifting from static read-only pages to highly dynamic services providing rich user interactions.

Nowadays, web applications are composed of three major components: (1) the client side or front end, running on the browser, (2) the server side, or back end, running on the server and (3) the data store, which also runs on a server (not necessarily the same as the back end) or on the cloud.

Framework. In web development, but more in general in computer programming, it is common to build an application upon a *software framework*. A software framework is an abstraction in

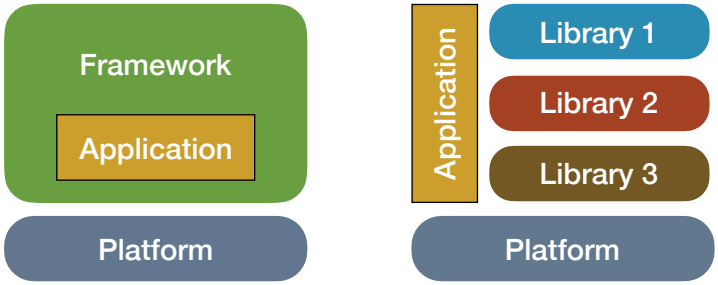


Figure 2.1: Library vs Framework comparison. Taken from [lib10]

which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific functionalities [1].

Frameworks and libraries, run on platforms. A platform is some lowest level on which to build an application, as operating systems (e.g. Linux, Windows, OSX) or virtual machines (e.g. Java, .Net). However, there are some differences between libraries and frameworks as Figure 2.1 shows. When an application is built using libraries, the application is in control. In other words, the application stands on its own, it has an identity, and it just uses libraries to do some part of the work. When building an application using a particular framework, the application lives inside the framework which controls the application. Practically, the framework embodies some abstract design and, in order to use it, the developer of the application needs to insert specific behaviours in various places of the framework (e.g. subclassing). The framework's code then calls the application code at these points [Inv05]. Such control switch is called *Inversion of Control* (IoC) and Martin Folwer describes it as a key part of what makes a framework different than a library.

Frameworks create a structured and organized environment in which a new application can be developed. Frameworks take care of low level details of the environment (e.g. rendering data on the browser) and let developers focus on the application specific behaviours.This greatly decreases the time needed to start creating applications and websites since developers do not have to start from scratch but they can reuse the framework's code. However, frameworks have also some limitations. For example, a framework can have a lot of unnecessary code that adds to the application's overhead and the scope of the framework can be binding in some cases as the application exists inside the framework.

2.1.1.1 The Client

For client side programming, even though developers often adopt a rich set of technologies and frameworks, they still mostly use the core stack of *HTML* plus *CSS* and *JavaScript*, with a few

[1] `https://en.wikipedia.org/wiki/Software_framework`

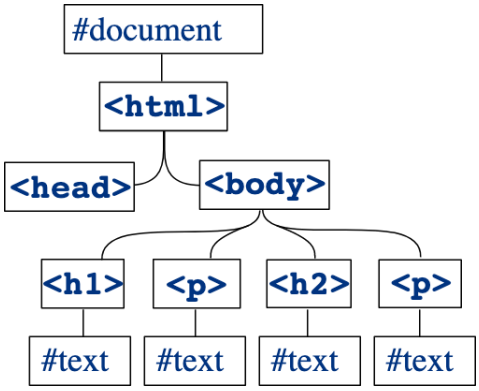


Figure 2.2: DOM tree example

extensions mostly provided through additional standards. The major reason for the success of such stack is the separation of structure/content (HTML), presentation/style (CSS) and logic (JavaScript), that is supposed to make reuse and maintainability much simpler.

HTML. HTML, which stands for *HyperText Markup Language*, is a technology that allows to specify the structure of the visual elements (sometimes referred to as the user interface) of a web application. An HTML document consists of normal text content wrapped around *tags*, which are a form of metadata that is used to apply structure to the content of the page. HTML is a hierarchical method of structuring documents by indenting HTML tags that are contained in other HTML tags. The hierarchical structure defined by HTML tags is called the *Document Object Model*, or *DOM* for short, which is used by the browser to create a visual rendering of the page. The DOM is a way of representing objects that can be defined via HTML and then later interacted with via a scripting language like JavaScript. HTML tags define DOM elements, which are entities that live in the DOM.

One way to visualize the DOM is by using tree diagrams shown in Figure 2.2. DOM elements that are lower in the tree are called *descendants* of DOM elements that are higher in the tree if there is a path that connects them. Immediate descendants are called *child elements*, and the element above a child element is referred to as its *parent element*.

CSS. CSS stands for *Cascading Style Sheets* and its main purpose is to give style to the content structured with HTML. Essentially, CSS files describes how specific elements of the HTML should be displayed by the browser. A CSS file is a collection of *rulesets* and a ruleset is a collection of style rules that are applied to some properly selected elements in the DOM. A ruleset consists of a selector and a list of rules, which are key-value pairs. One major objective achieved by means of CSS is *responsive* design. A design is said to be responsive if it changes its layout based on the height and width of the browser in which it is displayed (e.g. desktop, tablet, mobile devices). It is possible to obtain responsiveness using CSS media queries. However,

several CSS frameworks are available that allow developers to build responsive designs with less effort.

JavaScript. JavaScript is a cross-platform, object-oriented scripting language used to make websites interactive (e.g. having complex animations, clickable entities, popup menus, etc.). In the host environment, such as a web browser, JavaScript can be connected to the objects of its environment to provide programmatic control over them. Specifically, client side JavaScript can control the page displayed by a browser and its Document Object Model (DOM), for example, by dynamically placing elements on an HTML form and by responding to user events such as mouse clicks, form (e.g. key press), and page navigation.

Moreover, the browser environment provides other functionalities built on top of the core JavaScript language, in the form of *Application Programming Interfaces* (APIs). They generally fall into two categories: browser APIs and third party APIs. Browser APIs are built into the web browser and are able to expose data from the surrounding computer environment, for example retrieving geographical information (Geolocation API), creating animated 2D and 3D graphics (Canvas and WebGL API) and interacting with multimedia (Audio and Video APIs) for playing audio and video in a web page. Third party APIs are not built into the browser by default but they have to be imported in the web page as JavaScript libraries. They can provide functionalities of third party websites to the website under development, e.g. by embedding custom maps into it (Google Maps).

One interesting browser API is the DOM API that allows JavaScript to manipulate HTML and CSS at runtime, creating, removing and changing HTML, dynamically applying new styles to the web page, etc. Part of this API provides web *events*. In general, events are actions that happen in the system being programmed. The system will fire a signal of some kind when an event occurs, and also provide a mechanism by which a response can be automatically activated (e.g. some code running) when the event occurs. In the case of the web, events are fired inside the browser window, and are usually attached to a specific item that resides in it. Examples of web events are: the user clicking on an element, the user submitting a form, a web page finishing loading, etc. Each event has an *event listener* that listens out for the event to happen, and an *event handler* (or *callback*) which is the code that runs in response to the event being fired. Event handlers are sometimes called event listeners although, strictly speaking, they work together.

The JavaScript runtime uses a message queue to handle events.The message queue contains the list of messages to be processed. Indeed, in web browsers, messages are added anytime an event occurs if there is an event listener attached to it. Each message may have an associated function which gets called in order to handle the message, namely the event handler. The messages in the queue are processed in an *event loop*, starting with the oldest one. A very interesting property of the event loop model is that JavaScript never blocks. While the application is waiting for an event to be processed, JavaScript can still handle other user inputs.

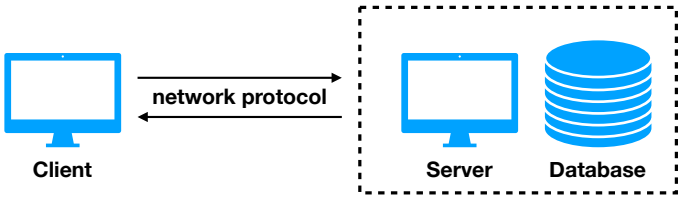


Figure 2.3: Client-server model

Usually, when a developer builds a new web application, almost surely, he/she will use a JavaScript framework, instead of writing vanilla (i.e. plain) JavaScript. Choosing a JavaScript framework, however, is not easy nowadays, given their proliferation. According to a recent survey [2] there are at least 57 JavaScript frameworks as of January 2018.

2.1.1.2 The Server

Traditionally, a *server* program abstracts some resource over a network that multiple *client* programs want to access. The simplest client-server model is shown in Figure 2.3.

In the context of web applications the client is a web browser and the server is a remote machine that is abstracting resources via the *HyperText Transfer Protocol*, or *HTTP* for short. Although it was originally designed to transfer HTML documents between computers, the HTTP protocol can now be used to abstract many different types of resources on a remote computer (for instance, documents, databases, or any other type of storage). HTTP is the basic technology behind the World Wide Web. Examples of HTTP servers are Apache or Nginx, designed to host big websites. An HTTP server is also event-driven but instead of being called in response of user events, it is called whenever a client (in our case, the browser) sends an HTTP request.

There are essentially four types of HTTP requests, also called *HTTP verbs*, namely *POST*, *GET*, *PUT* and *DELETE*. These four verbs map to four basic operations on the data store (persistent storage), respectively *Create*, *Read*, *Update* and *Delete*, also known under the acronym of *CRUD*. This mapping allows web developers to create APIs that create a clean and simple interface to resources that are available on the server (data store). APIs that behave in this way are typically referred to as *RESTful* web APIs. *REST* stands for *REpresentational State Transfer*, an architectural style of the web that describes how resources on web servers should be exposed via the HTTP protocol, first proposed by Roy Fielding [FT00]. REST specifies a layered client-stateless-server architecture in which each request is independent of the previous ones.

In the context of the back end, frameworks are also widely used. On Github [3], server-side frameworks count 29 repositories spanning 11 languages (last update March 2017).

[2] `https://jsreport.io/the-ultimate-guide-to-javascript-frameworks/`
[3] `https://github.com/showcases/web-application-frameworks`

2.1.1.3 The Data Store

The data store application program runs independently of the web application server and is responsible to persistently store the data of the web application. *SQL* (sometimes pronounced *sequel*), which stands for *Structured Query Language*, is probably the most popular technology that is used to interact with a database. SQL is a language that is used to perform queries on a database stored in a relational format. Relational databases store data as cells in tables, where the rows of the tables can easily be cross-referenced with other tables. For instance, a relational database might store a table of actors and actresses and a separate table of movies. A relational table can be used to map actors and actresses to the movies in which they star. This approach has several advantages, but the primary reason why they are so widely used is supposedly because storing information in this way minimizes redundancy (duplicated data) at the expense of some performance penalty.

The alternative consists of storing data in a non-relational format. Such new data storage solutions, sometimes referred to as *NoSQL* data stores, make a trade-off: sometimes they store redundant information in exchange for increased ease-of-use from a programming perspective. Furthermore, some of these data stores are designed with specific use cases in mind, such as applications where reading data needs to be more efficient than writing data. For example, let us consider a web application that shows restaurants in a user-specified town. The user can sign up and rate a restaurant he/she has been to. In such web application querying restaurants, together with their own reviews, is much more frequent than a user writing and submitting a review. Therefore, it is reasonable to store a restaurant together with its reviews even if they are different concepts and, in a traditional relational approach, they would probably be stored separately (i.e. different tables).

Regarding database technologies, 350 database management systems have been ranked [4]. Most of them fall into the categories of relational and non-relational (document) data storage.

2.1.1.4 Communication Between Client and Server

Today's practice for the communication between client and server consists of two elements. A standard format for the data being transmitted from client to server and viceversa, and a technology that allows asynchronous exchange of data.

JSON. *JSON*, which stands for *JavaScript Object Notation*, is the most widely used data-interchange format. A JSON object is a collection of variables each of which has a name and a value. The name is a string and the value can be either a string, a number, a boolean, an array or an object itself.

[4] `https://db-engines.com/en/ranking`

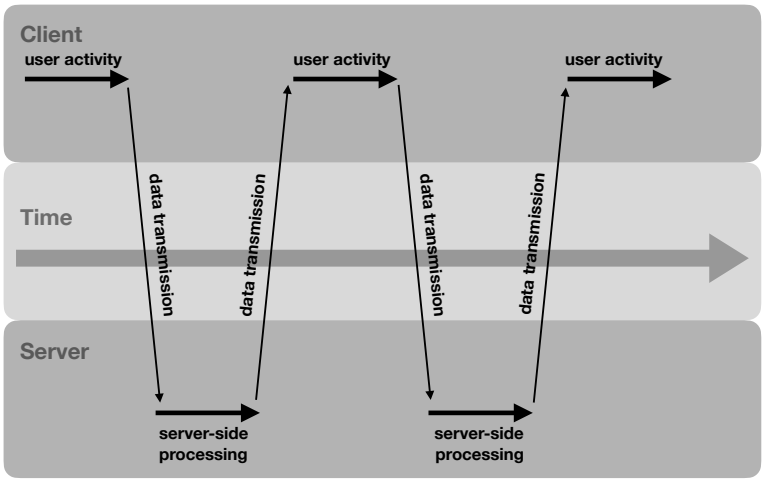


Figure 2.4: Multi Page Web Application Model. Taken from [aja05]

AJAX. *AJAX* stands for *Asynchrnonous JavaScript And XML*. The common data-interchange format that preceded JSON was called *XML*, which looks much more like HTML. Although XML is still widely used in many applications, there has been a major move to JSON since AJAX was invented. The reason is that JSON is as expressive as XML and, in addition, JSON can use arrays [jso19]. Furthermore JSON is shorter and quicker to read and write, since it is parsed into a ready-to-use JavaScript object, whereas XML requires a XML parser. The basic idea behind AJAX is that the web application can send and receive information to/from other computers without reloading the entire web page. One of the first examples of use of AJAX is Google's Gmail web app. A new email just appears in the inbox of Gmail's web page which does not need to be reloaded to get new messages.

The advent of AJAX changed the traditional synchronous communication model between client and server (shown in Figure 2.4), i.e. the so called *click and wait* model, introducing asynchronicity on the client side (shifting the communication model to the one shown in Figure 2.5). Nonetheless, the current web application development scene can still be divided according to two models, called *multi page application* and *single page application*.

Multi Page Applications. Multi page applications (MPAs) are the traditional web applications that reload the entire page and display the new one whenever a user interacts with the web application. The synchronous model of multi page web applications is shown in Figure 2.4. Whenever data is exchanged back and forth, a new page is requested from the server to be displayed in the web browser. This process takes time to generate the page on the server, to send it to the client and to display in the browser, which may affect negatively the user experience (the client is *frozen* while waiting for the new web page to be sent from the server and displayed).

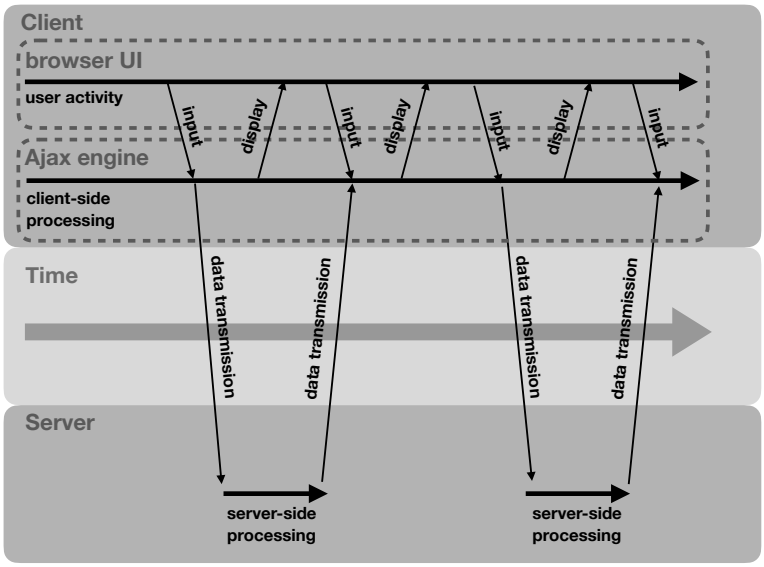


Figure 2.5: Single Page Web Application Model. Taken from [aja05]

AJAX has made it possible to render/update just a fragment of the web page, at the cost of more difficult and complex development process.

Single Page Applications. Single page applications (SPAs) consist of just one single web page. SPAs load all the content onto just one single page, which is updated dynamically, rather than navigating the user to different pages. SPAs are faster and provide a better user experience than multi page web applications because they execute a substantial portion of the application logic in the web browser itself rather than on the server. After the initial page load, when a web page with the client side code is sent to the client, only data is exchanged between client and server (usually in JSON format) instead of entire HTML pages. The asynchronous model of single page web applications is shown in Figure 2.5.

Despite the disadvantages of multi page applications in terms of development, maintainability, user experience and performance, MPAs still perform better than SPAs in terms of search engine indexing. This is one of the main reasons why MPAs are still widely popular and hybrid approaches, which try to combine the best of the two models, are also arising.

2.1.2 Software Testing

Software testing is widely used in the industry as a quality assurance technique for the various artefacts produced in a software project, including the specification, the design, and the source code. Testing aims at finding as many errors as possible, thus improving the reliability, the qual-

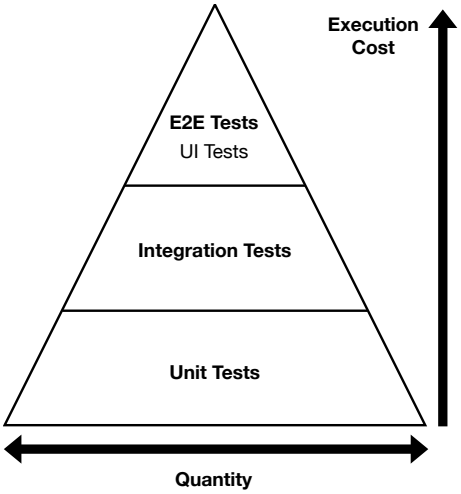


Figure 2.6: Testing pyramid taken from [Coh10]

ity and the compliance with expected behaviour of the software. The goal of testing, however, is not to prove correctness, because exhaustive testing is economically and technologically infeasible, even on trivial examples. Rather, it aims at increasing confidence on the software quality by showing that faults are not observed in a set of test scenarios that exercise the application according to some *test adequacy criterion*. Indeed, since testing resources are limited and there is no way to determine if all faults have been exposed, testers use an adequacy criterion as an indicator of sufficient testing to stop writing and executing new test cases. The test adequacy criterion usually consists of measuring some *coverage* of the system under test (SUT), while the system is being exercised by the test cases. For example, if the code of the SUT is available, structural coverage can be measured. Examples of structural coverage are statement and branch coverage. Instead, if the test cases are based on a finite state model that describes the functional aspects of the SUT, one common criterion is to measure the coverage of the transitions that constitute the model.

Another important question software testers are expected to answer is what to test. The testing pyramid shown in Figure 2.6, first introduced by Mike Cohn [Coh10], briefly describes what to test and to what extent. Indeed, the size of each layer indicates the number of tests that should be written within each stage. The first stage is constituted by *unit tests*, in which each unit is tested in isolation, after properly *mocking* all its dependencies. What a unit is depends on the program under test and on the level of granularity the tester chooses. For example, in object oriented programs, a unit is often a class. For its functioning such class may depend on several others but, since the objective is to test that class in isolation, the other classes the class under test depends on should be replaced with *mocks* or *stubs*, i.e. ad hoc classes that simulate the behaviours of real classes. The pyramid suggests that unit tests are cheap to execute and that the vast majority of tests in a software system should be unit tests. The second stage is given by *integration tests*

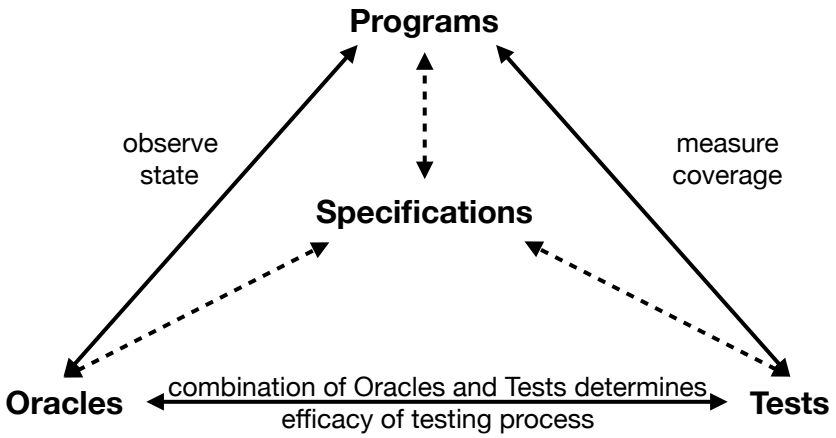


Figure 2.7: Theoretical testing framework: example relationships between testing factors. Taken from [SWH11]

in which the individual units are combined and tested as a group. If the unit is a class, then one possible criterion is to test together (i.e. without mocking) such class and all the classes the class under test depends on. According to the test pyramid integration tests are more expensive to execute than unit tests and there can be less integration tests than unit tests in the test suite of a software project. The last stage consists of *end to end tests*, E2E tests for short. Such tests exercise the system as a whole, under real world scenarios including the communications of the software under test with hardware, network, database and other programs. The system is tested in an E2E way through its user interface (that is the reason why they are often called *UI* tests), which is typically a *Graphical* user interface (GUI). Those tests are usually computationally more expensive than integration and unit tests, because they involve the entire system with all its components. That is why the overall percentage of such tests in a software system is usually low with respect to the other types of tests.

A more formal view of testing is shown in Figure 2.7. In such theoretical testing framework, Staats at al. [SWH11] illustrate the interrelationships between the four key testing artefacts: specifications, programs, tests and oracles. With the term specifications they mean the abstract, ideal notion of correctness (ground truth). Specifications in this context are the true requirements of the software, i.e. they represent the intended/desired behaviour of the software. The dashed arrows in Figure 2.7 represent approximations, i.e. possible sources of errors or divergences that make software testing needed and, at the same time, challenging. The first source of error is when the program is derived from, and intended to implement, the specification, for instance through requirements. Tests are, in a similar way, approximately derived from the specification and are intended to exercise scenarios where the program violates the specification. Finally, the

oracle, directly approximates the specification and it is intended to determine, for each test, if the program has violated the corresponding specification.

The other arrows of the diagram represent relationships between the concepts that can be explained as follows. Tests are selected based on the syntactic structure of the program under test, for example based on how well they cover the structural properties of the program (test adequacy criteria). The relationship between the oracle and the program under test implies the observability of the state of the program under test, i.e. weather it is possible to observe program failures and to identify program locations in which failures can be exposed. Finally, the relationship between oracles and tests is crucial for the efficacy of the testing process, which is failure exposure and fault finding.

2.1.3 Metaheuristic Algorithms

Stochastic optimization is the general class of algorithms and techniques which employ some degree of randomness to find optimal (or as optimal as possible) solutions to hard problems. *Metaheuristics* are the most general of these kinds of algorithms. Metaheuristics are applied to a very wide range of problems [Luk13]. In particular, they are applied to problems where it is possible to determine the solution analytically nor what is the direction towards it and exhaustive search cannot be applied because the solution space is too large. The assumption of metaheuristic algorithms is that, given a candidate solution, it is possible to assess how good that solution is. The quality of a solution is measured by the so called *fitness function* which has to be defined for each specific optimization problem. Furthermore, the candidate solution has to be *encoded* for each specific problem. One simple and common *representation* for candidate solutions is a fixed-length vector with boolean values.

Metaheuristic algorithms are divided into two classes: non-population based and population-based algorithms. The difference is in the number of candidate solutions: non-population based methods keep a single candidate solution, whereas, population based methods keep a sample of candidate solutions. Below, a class of population based algorithms (namely, *evolutionary algorithms*) is briefly explained.

The basic generational evolutionary computation algorithm (shown in Figure 2.8) first constructs an *initial* population, then iterates through three procedures. First, it *assesses* the fitness of all the individuals in the population. Second, it uses this fitness information to *breed* a new population of children. Third, it *joins* the parents and children in some way to form a new next-generation population, and the cycle continues, until a certain stopping condition is met (e.g. the optimal solution is found or the algorithm runs out of resources, like max time or max number of fitness evaluations). Evolutionary algorithms differ from one another largely in how they perform the *breed* and *join* operations. The breed operation usually has two parts: *selecting* parents from the old population, then reproducing them (usually *mutating* or *recombining* them in some way)

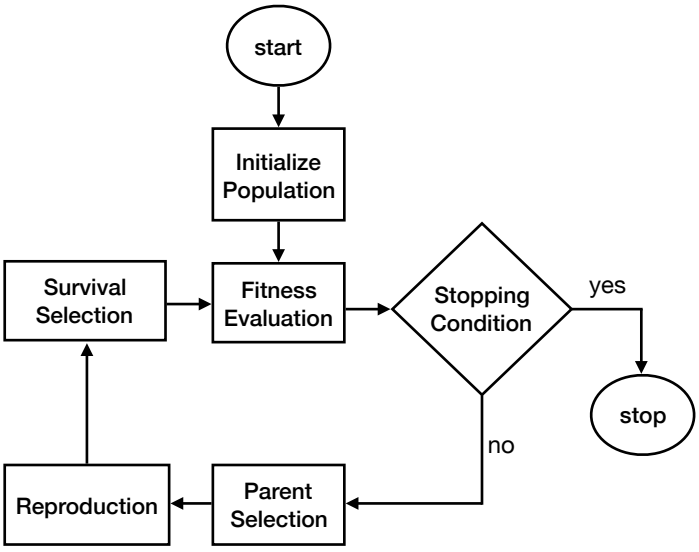


Figure 2.8: Relationships between components of an evolutionary algorithm. Taken from [ES+03].

to make children. The join operation either completely replaces the parents with the children, or includes fit parents along with their children to form the next generation. The *reproduction* block includes recombination (often called *crossover*) and mutation. The selection from the old population is called *parent selection*, whereas the join operation is called *survival selection*.

A common example of evolutionary algorithm is the *Genetic Algorithm* (GA), invented by John Holland in the 1970s. Basically, GA follows the steps described in Figure 2.8 [ES+03]. Given a population of n *individuals* (candidate solutions), parent selection fills an intermediate population of size n. Then the intermediate population is shuffled to create random pairs and crossover is applied to each consecutive pair with a certain probability and the children replace the parents immediately. Mutation is applied individual by individual to the new intermediate population (each individual is modified with independent probability). The resulting intermediate population forms the next generation, which replaces the previous one entirely.

2.2 Web Application Testing

In this thesis the focus is on E2E tests. Testing a web application E2E is important since today's web applications are built on intertwined layers of code, networks of subsystems and third-party integrations. This makes the stability of each component, both on its own and as part of the entire application system, vital to the functioning of a web application. Moreover, it highlights the clear

need to test entire web application from start to finish, from software modules and APIs through networking systems and end-user interfaces. Indeed, E2E tests have the advantage of exercising the web application with all its parts connected together and thus can find bugs in the interaction between components in the way that unit and integration tests cannot.

2.2.1 E2E Testing

In the context of web applications an E2E test exercises the application starting from the GUI but indirectly it exercises all the components that contribute to form the web application, back end and database server included.

2.2.1.1 Modelling

Formally, an E2E test case is a *triple* $T = \langle U, V, O \rangle$, where $U = \langle u_1, \ldots, u_n \rangle$ is a sequence of user events (e.g. click on a button/link, filling in a form, etc.), $V = \langle v_1, \ldots, v_n \rangle$ is the sequence of input values required by the user events (e.g. the values required for filling in a form), and $O = o_1 \wedge \ldots \wedge o_n$ is the oracle (expected results) for the sequence of user events. An item in the sequence V, i.e. v_i, can be empty (i.e. an empty sequence) which means that the corresponding user action u_i does not require any input value. The sequence of user events bring the application to a certain state, which is the state of the Document Object Model (DOM) which has been manipulated and modified as a result of the user events, plus the server side state. The oracle is supposed to verify the existence or properties of particular elements in the resulting DOM state. Therefore, each o_i is a function that takes l arguments, where $l \geq 0$, and returns a *boolean* value. If an o_i is not specified then its value is simply the boolean value `true`. Note that the function o_i can evaluate multiple conditions after the same step. If the oracle O, which is the conjunction of n functions, returns false, the test case fails and a *failure* is found, since the application under test deviates from the expected behaviour. Otherwise, if it returns true, the test case passes.

Figure 2.9 shows an example of an E2E test case that can be performed manually against the example application, called *Phoenix Trello* [5] which is the open source version of *Trello* [6], a web application that provides a visual way to manage and organize projects. Its main elements are boards, lists and cards that can be private to the single user or shared among users to enable collaboration.

The figure, from the left hand side to the right hand side, shows the transitions of the web application when the action at the top of each snapshot is performed. The E2E test case triple in this case is: $U = \langle$*open-app*, *insert*, *insert*, *click-sign-in*$\rangle$, $V = \langle\langle\rangle$, *"john@phoenix-trello.com"*, *"12345678"*, $\langle\rangle\rangle$ and $O =$ `true` $\wedge$ `true` $\wedge$ `true` $\wedge$ *check("Sign out")*. The E2E test case is testing the login

[5] `https://github.com/bigardone/phoenix-trello`
[6] `https://trello.com/`

Figure 2.9: Example of E2E test case

functionality of the web application, in the scenario in which the user who is signing in is already registered. The oracle at step (5), makes sure that the sequence of actions performed to sign in the user brings the application to the desired DOM state in which the *Sign out* text is present at the top right corner of the web page.

2.2.1.2 E2E Testing Tools

The test case presented in Figure 2.9 can be executed manually by carrying out the actions through the web browser. Manual testing is the simplest approach to testing but, unfortunately, this practice is error prone, time consuming, and ultimately not very effective. For this reason, most teams automate manual web testing by means of automated testing tools. The first part of the process is manual: the developer writes the test code to exercise the web application under test. The test code provides input data, operates on DOM elements, and retrieves information to

Table 2.1: Sample of E2E testing tools classified according to localization and test case development techniques. Taken from [LCRT16].

	Test Case Development	
Localization	Capture-Replay	Programmable
Visual-based	Sikuli IDE	Sikuli API
DOM-based	Selenium IDE	Selenium WebDriver

be compared with oracles (e.g., in the form of *assertions*). The second part of the process is automatic: the test code is executed automatically against the application under test. The execution is faster, with respect to the manual one and it can be carried out much more often.

E2E web testing tools can be classified according to two main orthogonal criteria [LCRT16]: (1) test cases implementation and (2) web page elements localization. Table 2.1 presents a sample existing E2E web testing tools classified according to these criteria. For what concerns the first criterion, there are two main approaches:

Capture-Replay (C&R) Web Testing. C&R consists of recording the actions performed manually by a tester on the web application GUI and producing a test case that automatically reproduce those actions.

Programmable Web Testing. It consists of full-fledged programs, written with the help of specific testing frameworks that interact with the web browser such that the test case can navigate the application under test as a user would (e.g. clicking on buttons and links, submitting forms, etc.).

An E2E test case that executes automatically can interact with web elements, such as links, buttons, forms, input fields, etc. The interaction with those elements require their *localization* in the web page. Correspondingly, for the second criterion (localization) there are two main approaches:

DOM-based. In this category elements are located using the information available in the DOM (see Section 2.1). The most prominent tools in this category are *Selenium IDE* [7] and *Selenium WebDriver* [8] respectively under the *Capture-Replay* and *Programmable* approach. Those tools offer the possibility to locate web elements in the DOM in different ways, e.g. by attributes like `name`, `id` and `class`, by `XPath` and by `CSS` selectors, by `tag name` and by `href`, which is the string *URL* that can be associated with a link element. However, not all location mechanisms can always be used to uniquely locate a web element in the DOM. For example a web element

[7] `https://www.seleniumhq.org/selenium-ide/`

[8] `https://www.seleniumhq.org/projects/webdriver/`

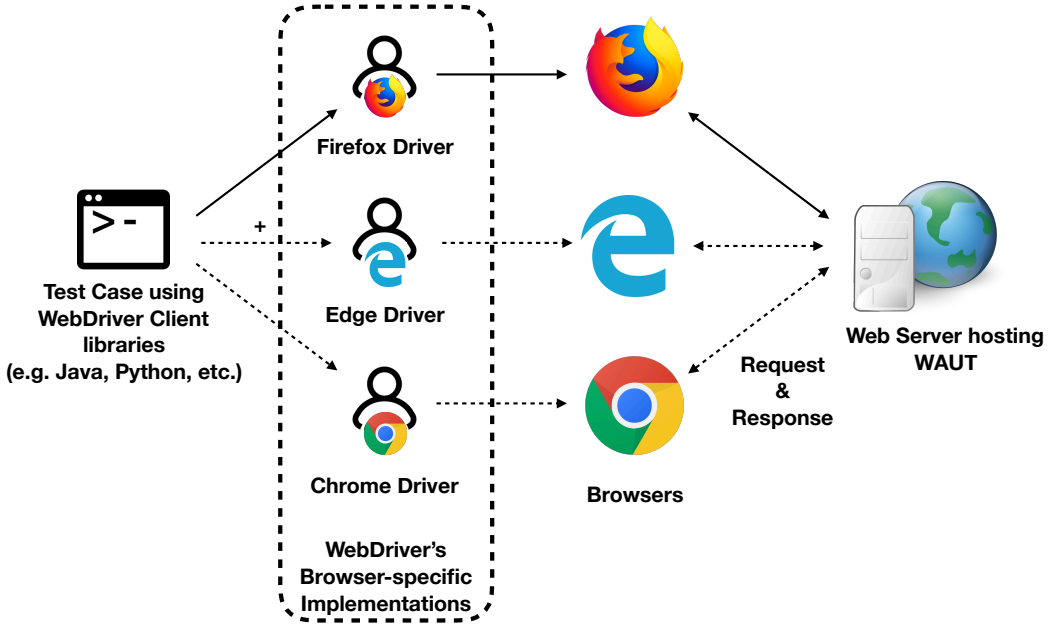


Figure 2.10: Selenium architecture. Taken from [Ava14].

can be located by attribute if it either has a unique `name`, `id` or `class`. In the same way a web element can be located by `tag name` only if there is only that element with that tag and it can be located by `href` if the *URL* attached to that link element uniquely identifies it. On the other hand, localization either through `XPath` or `CSS` can always be applied. Indeed, both those location techniques are hierarchical, meaning that it is always possible to uniquely identify a web element X by traversing the DOM tree starting from the root element R.

Visual-based. These tools exploit image recognition techniques to locate and control web elements (GUI elements). The tools Sikuli IDE [9] and Sikuli API [10] belong to this category, respectively for *Capture-Replay* and *Programmable* approach.

2.2.1.3 Selenium WebDriver

In this thesis, the focus is on *Selenium WebDriver*, i.e. the programmable DOM-based approach. *Selenium WebDriver* is a quite mature (it started in 2009) and well maintained tool (on the official GitHub repository hosting the project [11], in the month between August and September 2019, 82 commits have been pushed to the master branch and 84 commits to all branches) which was specifically designed to automate the actions performed on a web page programmatically.

[9] `http://www.sikuli.org/`
[10] `https://github.com/sikuli/sikuli-api`
[11] `https://github.com/SeleniumHQ/selenium`

Moreover, Selenium specifically provides infrastructure for the WebDriver specification, a platform and language-neutral coding interface compatible with all major web browsers. Since June 2018 [12] WebDriver became a W3C (World Wide Web Consortium, an international community that develops open standards) recommendation.

Overall, Selenium WebDriver works in the following way (see Figure 2.10 for a graphical representation) [Ava14]:

- A developer, through his/her test case, can command WebDriver to perform certain actions on the Web Application Under Test (WAUT) on a certain browser. The way the user can command WebDriver to perform such actions is by using the client libraries or language bindings provided by WebDriver. These libraries are available for different languages, such as Java, Ruby, Python, Perl, PHP, and .NET;
- By using the language-binding client libraries, developers can invoke the browser-specific implementations of WebDriver, such as Firefox Driver, Edge Driver, Chrome Driver, to interact with the WAUT on the respective browser. These browser-specific implementations of WebDriver work with the browser natively and execute commands from outside the browser to simulate exactly what a real user does. Language-binding client libraries supply a well-designed object-oriented API to let the test case interact with the selected browser;
- After execution, WebDriver sends out the test result back to the test case for developer's analysis.

Figure 2.11 shows the Selenium WebDriver version of the login test case for the application *Phoenix Trello*, previously shown in Figure 2.9. The figure presents at the top the login test case, implemented using the *Java* programming language and the Selenium WebDriver Java bindings. In the middle of the figure are the two screenshots of the WAUT before (left hand side) and after (right hand side) login is performed. Under each screenshot there is a portion of the DOM of the respective web page. The elements of the DOM the test case interacts with are highlighted. First, the test instantiates the specific selenium driver, in this case the `ChromeDriver`. Then it opens the browser at the application URL. The first driver statement is at line 5: it locates the email input field of the sign in form by its `id = "user_email"`; it then writes the email in the respective input field by using the selenium method `sendKeys`. At line 6, the driver statement is quite similar. Indeed, it locates the password input field by its `id = "user_password"` and it writes the password. At line 7, the test locates the submit button of the form by `XPath`, which is a *relative* `XPath` (it starts with a double slash), and it clicks on it. Such relative `XPath`, starting from the root, searches the first DOM element with the `button` tag which has an attribute named `type` with value `submit`. The assertion at line 8 is the oracle, which checks that the

[12] `https://www.w3.org/TR/webdriver/`

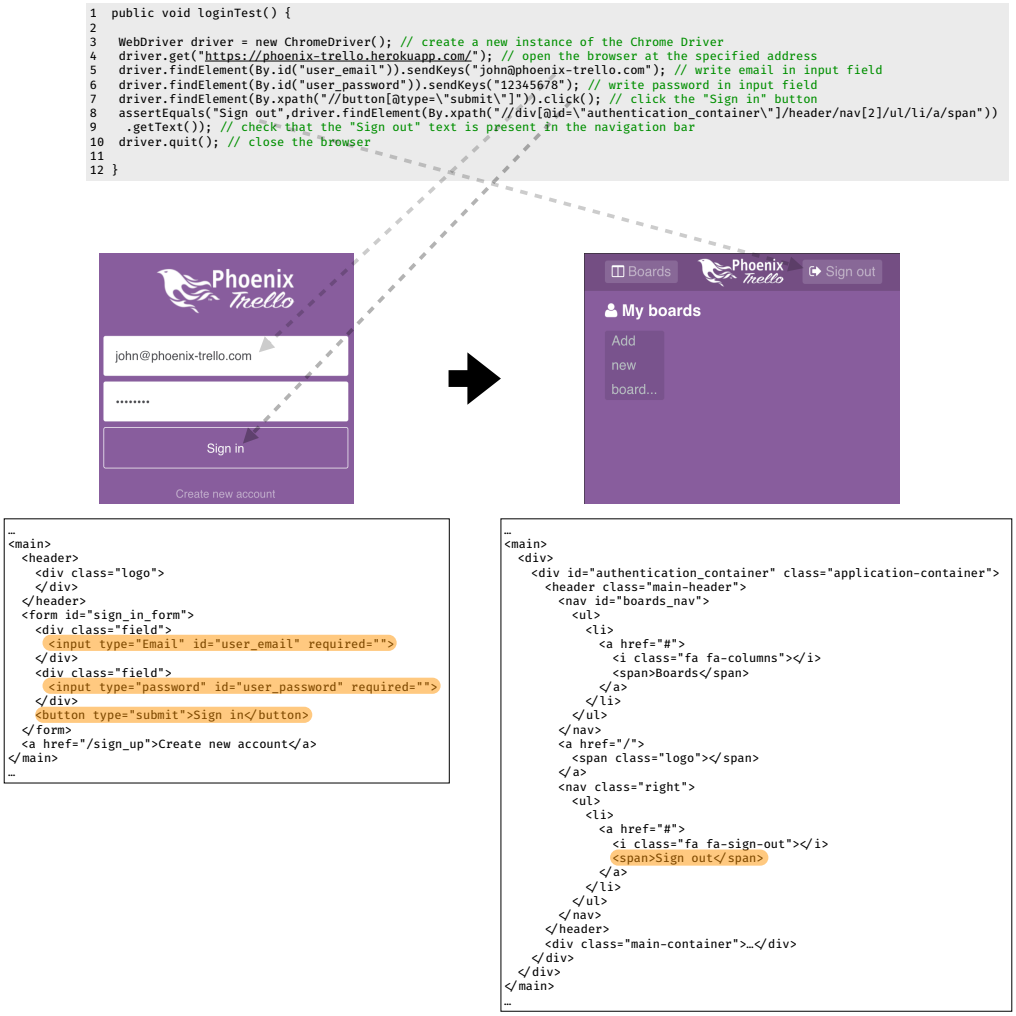


Figure 2.11: Selenium WebDriver login test case for Phoenix Trello application

login has been performed correctly. Also in this case the DOM element is located by a relative `XPath`. The `XPath` starts with a `div` which has `id = "authentication_container"` and it navigates the DOM, starting from that element, all the way through the element which has the `span` tag. The `getText` method of the driver extracts the text from a DOM element, in this case the text `Sign out`. The `assertEquals(<expected>,<actual>)` method of *JUnit* [13], a Java library for unit testing, checks that the first argument is equal to the second one. At line 8, it is used to check that the `Sign out` string is equal to the text extracted from the DOM element that is supposed to contain such text. If the two arguments are equal, the assertion

[13] `https://junit.org/junit5/`

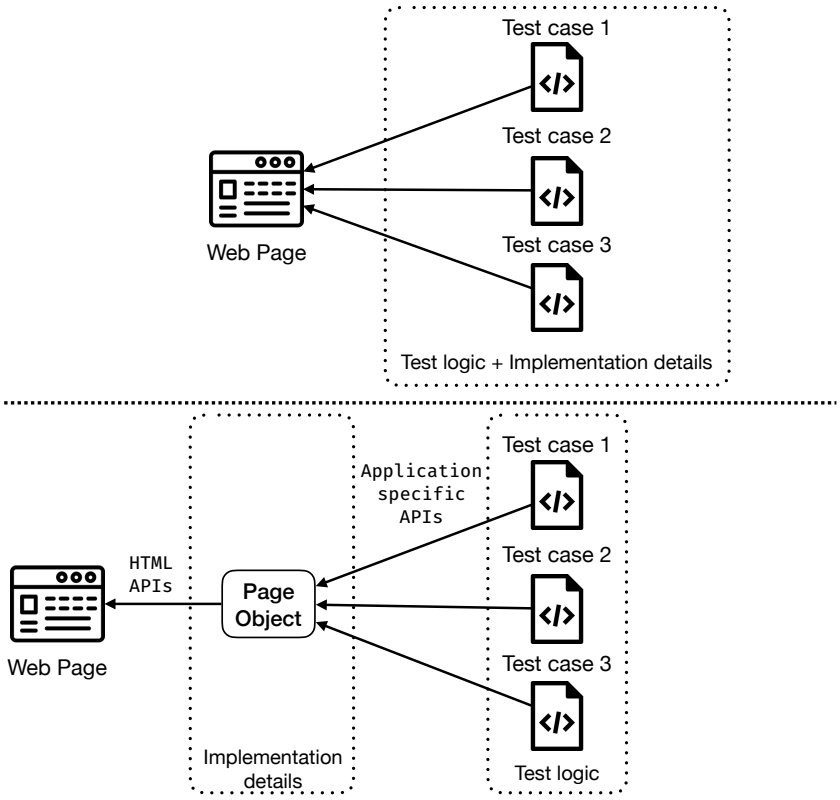


Figure 2.12: Test cases interacting with a web page without (above) and with (below) the use of the Page Object pattern

passes (hence, the test case); otherwise it throws an assertion exception, making the test case fail. Finally, the driver statement at line 10 closes the driver and ends the test case.

Page Object Design Pattern. The login test case in Figure 2.11 interacts directly with the web pages before and after the login is performed. Some of the implementation details of the web pages are included in the test case as locators. Indeed, the driver statements, from line 5 to line 8, depend on the structure (see `XPath` at line 7 and 8) and the properties (see `id` at line 5 and 6) of the web elements to be located and interacted with. Moreover, the same test case contains the test logic/semantic, i.e. what the test case actually does, which is logging in a user. The upper part of Figure 2.12 shows, in general, such testing scenario.

The fact that the test case contains both test logic and implementation details (locators) raises several software engineering issues such as *maintainability*, *reusability* and *readability* of test code. Regarding readability, the test logic is not completely evident by reading the instructions of the test case, which are selenium specific and implementation related (e.g. `findElement`, `sendKeys`, `click`, etc.). Regarding reusability, if another test case (e.g. `addBoardTest`) needs to perform the login before testing the functionality it was created for, it has to *duplicate*

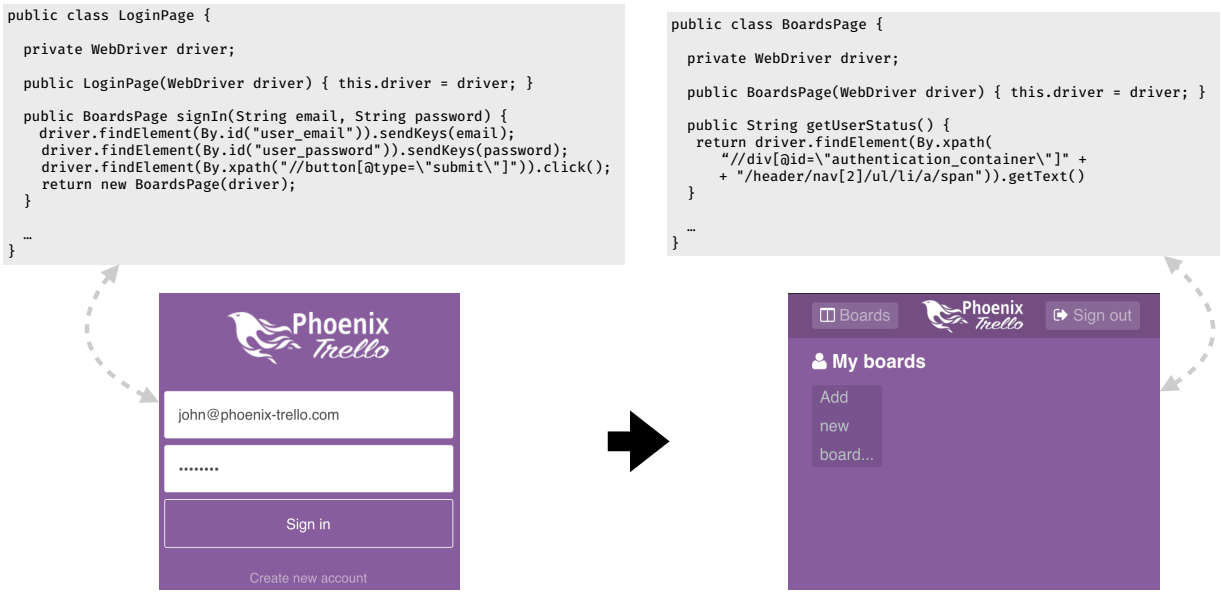
the specific selenium instructions from line 5 to line 7. Duplication has also a negative impact when it comes to maintainability, i.e. when the application under test evolves and the test cases need to be updated. Suppose that the `ids` of the email and password input fields of the login page (see screenshot at left hand side of Figure 2.11) change. Technically, such locators are called *broken*, since executing the test case on the new version of the application leads the test case to break. As a consequence, all the test cases that contain such `ids` need to be modified, which is an error-prone and costly process.

One way to alleviate this problem is through the use of the *Page Object Desing Pattern*, which is widely used in web testing to decouple the implementation details that depend on the web page structure from the test logic. The lower part of Figure 2.12 shows how test cases interact with the web page when the Page Object Design Pattern is adopted. The Page Object (PO for short) design pattern was first proposed by Martin Fowler [14] as an abstraction of the page under test.

POs are particularly important to increase the *maintainability* and the evolution of web test cases [LCRT16], since one of the main causes of test case breakages upon software evolution is the occurrence of broken locators [LSRT16]. If locators (implementation details) are confined within POs, they need to be maintained just once upon software evolution. When the concrete HTML page changes, those changes impact the POs, not the test cases, making the test cases more maintainable. Moreover, POs can be *reused* across test cases [LCRT16]. In fact, different test cases can refer to the same page object for locating and activating the HTML elements of the page under test, without having to duplicate the HTML access instructions multiple times, in different test cases. Finally, the PO pattern makes the test cases more *readable* since it creates an abstraction layer allowing the developer to write application specific APIs that are used by test cases. In summary, POs are a classic example of *encapsulation* applied to programmable web testing.

Figure 2.13 shows how the login test case in Figure 2.11 can be refactored using the PO design pattern. Two POs are created in this case for the two web pages showed in Figure 2.13. In the Java language they are just classes. The PO `LoginPage` (left hand side) refers to the form displayed to sign the user in; the second PO, `BoardsPage`, models the web page that displays the list of boards (right hand side) after the login. Another PO could be created in this case, i.e. the `NavigationBar` PO, which could model the functionalities of the navigation bar at the top of the web page on the right hand side. In particular, the board menu functionality on the top left corner, the application logo at the top center, which brings the application to the home page, and the sign out button on the top right corner. In fact, despite the term "page" object, these classes are not necessarily built for an entire page. A PO may wrap an entire HTML page or a cohesive fragment that performs a specific functionality. The rule of thumb is to group and model the functionalities offered by a page as they are perceived by the user of the application. The test case at the bottom of Figure 2.13 uses the POs `LoginPage` and `BoardsPage` to perform the same operation carried out in Figure 2.11, i.e. signing the user in and checking that

[14] `https://martinfowler.com/bliki/PageObject.html`




```
1  public void loginTest() {
2
3   WebDriver driver = new ChromeDriver(); // create a new instance of the Chrome Driver
4   driver.get("https://phoenix-trello.herokuapp.com/"); // open the browser at the specified address
5   LoginPage loginPage = new LoginPage(driver); // create a new instance of the LoginPage PO
6   BoardsPage boardsPage = loginPage.signIn("john@phoenix-trello.com", "12345678"); // sing the user in
8   assertEquals("Sign out",boardsPage.getUserStatus()); // check that the "Sign out" text is present in the navigation bar
9   driver.quit(); // close the browser
10
11 }
```

Figure 2.13: Selenium WebDriver login test case for Phoenix Trello application using Page Objects

the sign in succeeded. The method `signIn` of the `LoginPage` PO, fills in the login form with the concrete values of email and password passed to it in the test case and clicks on the submit button. The `signIn` method returns the PO that models the boards list since the click on the submit button of the login form leads the application to the boards list page. Then, the method `getUserStatus` of the PO `BoardsPage` is used in the assertion to retrieve the text at the top right corner of the boards list page, to be compared against the expected value. In this way the implementation details of the web page are confined inside the PO methods, which represent application/domain specific APIs, making the test case more maintainable and readable.

2.2.1.4 Pros and Cons of E2E Testing

To summarize the discussion on E2E web testing, below are highlighted the advantages and disadvantages of using this testing approach.

Advantages. The first advantage is that E2E testing *hides the technologies* that are used to build the web application under test, since the application is tested via its interface. The testing approach is general and it can be applied to any web application regardless of the technologies used to develop it. Moreover, an E2E test *exercises the application in real conditions/scenarios*, which helps developers increase their confidence on the quality of the application, since it navigates the application as a real user would. If a bug is introduced that hinders the core functionalities of the application, E2E tests can find it. Moreover, the *execution* of E2E tests can be *automated* by quite mature and well maintained tools like Selenium.

Disadvantages. On the other hand, E2E tests are *time consuming to run* with respect to the other types of tests, since the execution environment of those tests is the browser. This problem has been alleviated, in part, with the introduction of *headless* browsers, i.e. browsers that provide the same functionalities of normal browsers but without the user interface. Hence, the browser instantiation is faster with respect to the normal browser as well as the actions that are performed by robots (such as Selenium) against the web application under test. However, an E2E test case still remains slower than a unit test case [15] [16].

Moreover, E2E tests are notoriously *flaky* and often fail for unexpected and unforeseeable (non deterministic) reasons. The more sophisticated the user interface, the more flaky the tests tend to become. Timing issues, animations, unexpected popup dialogs, browser quirks are only some of the reasons for E2E test flakiness.

E2E tests are also prone to the so called *fragility problem*, i.e. minor changes to the web app under test can invalidate/break the existing test code. This problem is in part mitigated if E2E test cases are written with the Page Object design pattern that separates the implementation details from the test logic. Once such minor changes happen, only the specific page objects affected by them need to be modified, leaving the test cases unchanged.

Finally, when a failure occurs upon the execution of an E2E test, it is not possible to *pinpoint the root cause of the failure*, since anything in the entire flow could have contributed to raise the error.

2.2.2 Test Dependency

Ideally, running the tests in a test suite in any order should produce the same outcome [GBZ18]. This means tests should deterministically pass or fail *independently* from the order in which they are executed. A test dynamically alters the state of the WAUT in order to assess if it behaves as expected. In practice, some tests fail to undo their effects on the program's state after their execution, which can pollute any shared state [GSHM15] in tests executed subsequently.

[15] `https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html`

[16] `https://vuejsdevelopers.com/2019/04/01/vue-testing-unit-vs-e2e/`

Table 2.2: Test cases for Phoenix Trello, numbered according to their test execution order.

Test	Name	Description
t_1	`addUserTest`	A new user account is created.
t_2	`loginUserTest`	The newly created user logs in to the application.
t_3	`addBoardTest`	The admin adds a board to his/her board page.
t_4	`shareBoardTest`	The admin shares his/her board with the newly created user.

In the web domain, testers write E2E tests by using testing frameworks like Selenium WebDriver. Unlike unit testing, in which tests target specific class methods, web tests simulate E2E user scenarios, and therefore the program state that persists across test case executions might be left polluted, causing test failures if tests are reordered. This is due to the fact that the state of a web application is more distributed than the state of a single unit and, hence, more difficult to control and to clean up after test execution.

Table 2.2 lists four E2E Selenium WebDriver tests for the Phoenix Trello web application. Tests are numbered according to their test execution order. The execution of the first test `addUserTest` pollutes the state of the web application, which is used by the subsequent test `loginUserTest` to sign in the same user created by `addUserTest`. To make these two tests independent and avoid polluted program states, a tester, for instance, should (1) delete the user created in `addUserTest` *after* its execution, to clean the polluted program state, and (2) re-create the same user (or a different one) *before* the execution of `loginUserTest`.

In practice, however, testers re-use states created by preceding tests to avoid test redundancy, higher test maintenance cost and increased test execution time [LCRT16]. In doing so, they also enforce pre-defined test execution orders, which in turn inhibit utilizing test optimization techniques such as test prioritization [RUCH01].

Test Dependency Graph. Dependencies occurring between tests can be represented in a test dependency graph (TDG) [GBZ18], a directed acyclic graph in which nodes represent test cases and edges represent dependencies. TDG contains an edge from a test t_2 to a test t_1 if t_2 depends on t_1 for its execution (notationally, $t_2 \rightarrow t_1$).

Figure 2.14 illustrates the actual test dependency graph (TDG) for the test suite in Table 2.2. It contains an edge from `loginUserTest` to `addUserTest` because `loginUserTest` requires the execution of `addUserTest` to produce the expected result (i.e., `addUserTest` *must be executed before* `loginUserTest` in order for it to succeed). Multiple test dependencies can also occur. For instance, `shareBoardTest` depends both on `addUserTest` and `addBoardTest` for its correct execution, since the admin user, in order to share his/her board, needs both a board (t_3) and a user to share such board with (t_1).

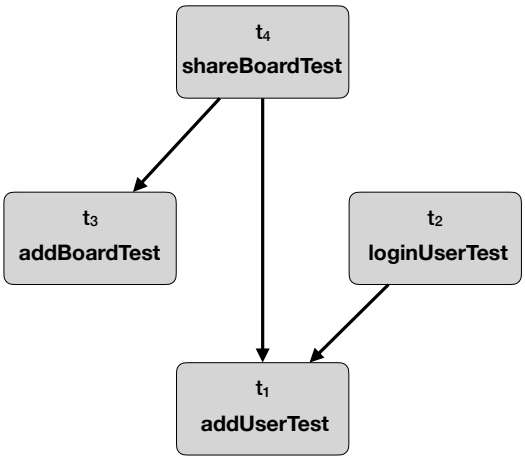


Figure 2.14: Test depencency graph for the test suite of Table 2.2.

2.3 Automatic Web Application Testing

As a bottom line, test automation means that the developers write the test cases and a machine/robot executes them. However, software testing researchers aim for a higher level of test automation. Indeed, their goal is that the machine both designs and executes the test cases.

Automatically generating an E2E test means generating all the components in the triple $T = \langle U, V, O \rangle$ presented in Section 2.2. Respectively, the machine should generate the sequence of user events U, the input values V that are needed by the user events and the oracle O. However, in this thesis, the automatic generation of the oracle is not addressed. Indeed the oracle problem is, in itself, a very challenging problem that would deserve other PhD theses. Instead, different techniques are explored in the subsequent chapters to address the problem of generating the sequence of user events U with the respective input values V. When the oracle is not explicitly defined in a test case, the test case relies on the so called *implicit* oracle. The implicit oracle catches a particular kind of unwanted behaviour of a program, which is the crash/error. In a Java program a failure may be an undeclared exception being thrown, whereas in a web application a failure may be represented by a client or a server error. In both cases, after the failure, the functioning of the WAUT is compromised or interrupted.

One way to generate the sequence of user events and their respective input values is through *model based testing*, which is presented below.

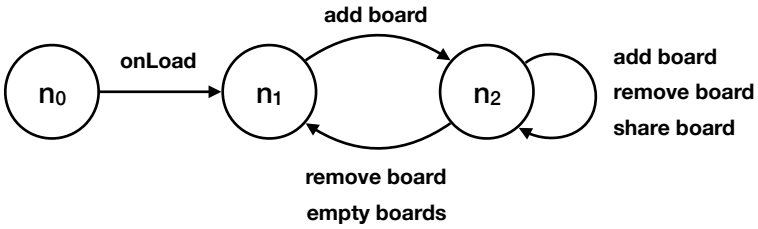


Figure 2.15: Abstract model of a simple Trello application

2.3.1 Model Based Testing

Model-based testing (MBT) is a variant of testing that relies on explicit behaviour models that encode the intended behaviours of a system under test (SUT) and/or the behaviour of its environment [UPL12]. Test cases are then derived from the model using extraction algorithms that try to satisfy some coverage criteria and are executed on the SUT. Hence, the SUT is seen as a black box that accepts inputs and produces outputs. The SUT has an internal state which changes as it processes inputs and produces output. The model describes possible input/output sequences at a chosen level of abstraction of the SUT's internal states.

In the context of web application testing, models represent an appealing solution, since they can abstract from the underneath technologies and they can capture all the important behaviours of the web application under test. Indeed a web application is nowadays developed using different programming languages that interact in a complex way to shape the overall behaviour of the application (see Section 2.1). Hence, it is difficult to base testing on the source code whereas models are a valuable alternative.

Usually, the model of a web application is a *state-flow* graph which captures the states of the WAUT, and the possible transitions between them. Nodes $\mathcal{N}$ of such graph are states of the WAUT and edges $\mathcal{E}$ are the events that bring the WAUT from one state to another. Basically, the model represents the navigations that a user can make while browsing the functionalities of the application (this is why the state-flow graph is often called *navigation* graph). A path in such graph can be defined as $P = \langle S, A, X \rangle$, where $S \in \mathcal{N}^+$ is a sequence of one or more graph nodes (states), $A \in \mathcal{E}^*$ is a sequence of zero or more graph edges (actions) such that $|A| = |S| - 1$ and if $a_i = \langle n_1, n_2 \rangle$ then $s_i = n_1 \in S$, $s_{i+1} = n_2 \in S$. A test case can be defined as $T = \langle S, A, X \rangle$ where, $X \in I^*$ is a sequence of zero or more parameter values matching the parameter names required by the events associated with A.

Figure 2.15 shows the model of a simple Trello web application with two states (n_0 is the initial state): n_1, in which there is no board in the project, and n_2, in which there is at least one board in the project. In this model all the important transitions of the application are represented: it is possible to add a board to the project, remove it, empty the project from all the boards and

share a board. Therefore, if testing is based on such model, with the test cases extracted from the model, all the important functionalities of the web application under test can be exercised. If the test adequacy criterion is transition coverage, testing based on the model in Figure 2.15 is considered sufficient when each transition is exercised at least once.

The general approach in model based testing is composed of the following steps, which can be applied to web applications as well:

1. **Getting the model**: available options are manual creation or automatic extraction;
2. **Using the model**: once the model is available, paths P, that represent actions on the application states, need to be generated. However, paths are abstract, since the model is abstract, hence, in order to make them executable against the WAUT concrete input data X need to be generated for the events in the sequences A;
3. **Checking path feasibility**: the last step consists of checking the absence of infeasible paths among the paths produced in the previous step. In fact, given the abstraction of the model there can be dependencies and constraints that hold in the WAUT but might not be represented in the model.

The test case feasibility problem is a major problem in model based testing [TTN14, DNSVT07], since it can impact the coverage of the model (e.g. transition coverage), the code coverage of the application under test and, consequently, the fault detection capability of the test cases. For example from the model in Figure 2.15, the following paths can be extracted: the first one is $P_1 = \langle S_1, A_1 \rangle$ where $S_1 = \langle n_1, n_2, n_1 \rangle$, $A_1 = \langle \textit{add}, \textit{remove} \rangle$; the second one is $P_2 = \langle S_2, A_2 \rangle$ where $S_2 = \langle n_1, n_2, n_2, n_2 \rangle$, $A_2 = \langle \textit{add}, \textit{remove}, \textit{share} \rangle$. The first is a feasible path whereas the second is an infeasible one, since the board that is added is necessarily the same that is removed afterwords. In such case the transition `share` is not covered, unless other paths are extracted. In a state-flow graph, transitions are always feasible, whereas paths extracted from the graph can be infeasible, as in the example shown above.

2.3.2 Web Crawling

One way to automatically get the model of a web application and use it is *web crawling*. A web crawler can automatically walk through different states of the WAUT and create a model of the navigational paths and states [MvDL12]. Crawling an SPA is more difficult than crawling a traditional MPA. In classical multi-page web applications, states are explicit, and correspond to pages that have a unique URL assigned to them. In SPAs, however, the state of the user interface is determined dynamically, through changes in the DOM that are only visible after executing the corresponding JavaScript code.

In order to explain how a crawler for SPAs works, we describe Crawljax, the first automatic approach that addressed the problem of SPA crawling.

2.3.2.1 Crawljax: Crawling Ajax Applications

The approach implemented in Crawljax [MvDL12] is based on a crawler that can exercise client side code, and can identify clickable elements (which may change with every click) that change the state within the browser's dynamically built DOM. From these state changes, a state-flow graph is inferred incrementally. Initially, it only contains the root state and new states are created and added as the application is crawled and state changes are analyzed. Below are the steps that the crawler follows to create such state-flow graph.

Detecting Clickables. The first operation that the crawler carries out is scanning the DOM of the current application state to analyze the clickable elements, which are those elements that respond to user actions (e.g. links, buttons) or, in other words, those that are exposed to an event type (e.g. click, mouseOver). For each candidate clickable element (candidate element for short), the crawler instructs a robot (e.g. Selenium WebDriver, used to simulate user input) to execute a click on the element (or perform other actions, e.g., mouseOver), in the browser.

Creating States. After firing an event on a candidate element, the algorithm compares the resulting DOM tree with the DOM tree as it was just before the event fired, in order to determine whether the event results in a state change. The problem of detecting already visited states is delegated to the *state abstraction function*, which is a function (which returns a value between 0.0 and 1.0) that decides if a new state is found after an event is fired. One way to determine the state change is by computing the Levenshtein *edit distance* [Lev66] between two DOM trees. A similarity threshold τ is used under which two DOM trees are considered clones. This threshold ranges from 0.0 to 1.0 and it can be set by the user of the crawler. A threshold of 0 means two DOM states are seen as clones if they are exactly the same in terms of structure and content. Any change is, therefore, seen as a state change. If a change is detected according to the distance threshold τ, a new state is created and added to the state-flow graph. Furthermore, a new edge is created on the graph between the state before the event and the current state.

Exploration Strategy. The crawler explores the web application according to the depth-first graph visit algorithm, meaning that, on each state, one clickable is selected, and fired an event upon. If such event reveals a new DOM state, the visit continues from the newly discovered state until no more candidate elements are present in the current state. In order to avoid an infinite loop, a list of visited candidate elements is maintained to exclude already checked elements in the recursive algorithm.

Backtracking. When no more candidate elements are present in the current state, the crawler backtracks its exploration to the first state containing unexplored candidate elements. One way to get to a previous state is by saving information about the clickable elements and the order in

which they have been fired. If such information is available, the application can be reloaded and the path from the initial state to the desired state can be followed by firing the clickable elements. As an optimization step, Dijkstra's shortest path algorithm [Dij59] can be used to find the shortest element execution path on the graph to a certain state. However, because of side effects of the clickable execution on the state of the WAUT, there is no guarantee that the exact same state is reached when a path is traversed a second time.

The state-flow graph obtained with Crawljax can be used for testing. In particular, it is possible to automatically generate test cases by extracting paths from the graph. One strategy is to apply graph visit algorithms (e.g. breadth-first and depth-first) that guarantee that each edge of the graph is included at least once. Each path is then made executable either by providing fresh new random inputs to the corresponding sequence of events in the path or by reusing the input values supplied during crawling. In this way, paths are extracted from the state-flow graph and, if all of them are feasible, the transition coverage test adequacy criterion is met. However, infeasibility is one of the major limitations of crawler-based approaches. State-of-the-art approaches to web testing and their limitations are discussed in the next chapter.

Chapter 3

State of the Art

This chapter presents an overview of the state of the art in the area of web testing. The chapter is organized into two main sections: we first discuss test generation techniques starting from those proposed for traditional software (e.g. Java) and then proceeding to state of the art approaches proposed in the literature for testing web applications. Then we present works related to test dependency including techniques for detecting test dependencies and test optimization techniques that assume test independence (e.g. test parallelization). We conclude the chapter by discussing the problem of test flakiness and how it is related to test dependency.

3.1 Test Case Generation

3.1.1 Traditional Software

In this section we present the most prominent techniques for automatic generation of test cases [CCC+13] for traditional software. In the next section we comment test generation approaches instantiated in the context of web applications.

Symbolic Execution. One widely used white box technique to generate test inputs for a program to improve code coverage and expose software bugs is *symbolic execution* [Kin75] (SE). SE uses symbolic values, instead of concrete values, as program inputs, and represents the values of program variables as symbolic expressions of the symbolic inputs. At any point during symbolic execution, this technique computes the symbolic values of program variables and a path constraint on the symbolic values to reach that point. The path constraint (PC) is a boolean formula over the symbolic inputs, defined by propagation of inputs along the path. Therefore, it represents the constraints that the inputs must satisfy in order for an execution to follow that path. If the PC is solved (using a constraint solver) it is called *satisfiable* and any solution of the PC is a program

input that executes the corresponding path which is a *feasible* program path. Otherwise the PC is called *unsatisfiable* and the corresponding program path is *infeasible*. Using SE for test input generation received much attention in the recent years [GKS05, SMA05, CGK+11, VPK04] both due to the dramatic growth in the computational power of today's computers and the availability of increasingly powerful constraint solvers (e.g. *Z3* [DMB08], *Yices* [DDM06], *STP* [GD07]). However, the effectiveness of symbolic execution on real world programs is still limited because the technique suffers from three fundamental problems: *path explosion*, *path divergence*, and *complex constraints*. Path explosion relates to the problem of the virtually infinite number of paths in the code. Path divergence is about the inability to compute precise path constraints due to multiple programming languages used to develop the program under analysis or parts of the program that may be available only in binary form (e.g. external libraries). Such inability leads to path divergence, i.e. the path that the program takes for the generated test data diverges form the path for which test data is generated. Moreover, some constraints may be complex, such that the general class of those constraints (e.g. constraints involving non-linear operations) is undecidable using the available constraint solver. These issues limit the applicability and the effectiveness of symbolic execution in generating test inputs for the program under test.

However, there are techniques that attempt to alleviate these three problems. For example, to partially address the complex constraints problem, two extensions of symbolic execution were proposed, respectively called *dynamic symbolic execution* (DSE) [GKS05] and *concolic execution* [SMA05]. Both DSE and concolic execution (CONCrete and symbOLIC) start from an available (random), concrete execution and use symbolic execution to explore alternative paths.

Random Testing. Random Testing (RT) is one of the most popular testing methods. It is simple in concept, easy to implement, and can be used on its own or as a component of many other testing methods. It may be the only practically feasible technique if the specifications are incomplete and the source code is unavailable. RT is often used in scientific papers as a baseline for comparison to evaluate a new test generation technique [AIB10].

Adaptive Random Testing (ART) [CLM04, CKMT10] has been proposed as an enhancement to RT. Several empirical studies have shown that failure-causing inputs tend to form contiguous failure regions, hence non-failure-causing inputs are also expected to form contiguous non-failure regions [WC80]. Therefore, if previous test cases have not revealed a failure, new test cases should be far away from the already executed non-failure-causing test cases. Hence, test cases are generated to be evenly spread across the input domain. It is this concept of even spreading of test cases across the input domain the basic intuition behind ART. Even spreading, diversity or distance between test cases (and test inputs), can be implemented in different ways and, for that reason, several algorithms have been proposed in the literature [CCT06, CLM04, CKL09, CKMN04, STM12]. All of them define methods to decide which test case to execute next, given a set of already executed test cases and an implementation of a distance measure between test cases. The process of adding test cases to the set of already executed test cases stops when a given criterion is met (e.g., maximum test suite size) or when a timeout is reached. Empiri-

cal studies [AB11, CLM04, CKM06] show that ART outperforms RT, in terms of effectiveness, considering both injected and real faults. In terms of efficiency all ART algorithms have an overhead compared to RT, given by the additional task of distance computation. Therefore, it is not obvious that an ART algorithm is more cost-effective than RT for a given program, where cost-effectiveness refers to the fault detection capability achieved in a given time budget.

Another important improvement of random testing is achieved by incorporating feedback obtained from executing test inputs as they are created [PLEB07]. The technique is called *feedback-directed random test generation* and it builds inputs incrementally using the feedback from previous executions to select those inputs that have the possibility to explore new program behaviours. In particular the tool Randoop [PLEB07] addresses random generation of unit tests for object oriented programs. Such kind of testing typically consists of a sequence of method calls, with the respective concrete input values, that create and manipulate the state of objects. The feedback from previous executions is used to avoid the generation of inputs leading to redundant or illegal (i.e. exception thrown) states, thus focusing the search towards unexplored object states.

Search Based Testing. Search Based Software Testing (SBST) is a branch of Search Based Software Engineering [HJ01] in which optimization algorithms are used to automate the search for test data that maximizes the achievement of test goals while minimizing test costs. In its simplest formulation the search problem consists of finding a value x^* which maximizes (or minimizes) the objective (fitness) function f over the search space X. In formulae: given $f : X \rightarrow \mathbb{R}$ the objective is to find $x^* : \forall x \in X, f(x^*) \geq f(x)$. The search is granted a search budget (e.g. time or max number of fitness evaluations) and it finishes when the search budget terminates or the testing adequacy criterion is met. However, given a search budget, search algorithms are only guaranteed to find a local optimum solution for a given problem.

In all approaches to SBST, the primary concern is to define a fitness function (or set of fitness functions) that capture the test objectives. The fitness function is used to guide a search based optimization algorithm, which searches the space of test inputs to find those that meet the test objectives. Because any test objective can, in principle, be re-cast as a fitness function, the approach is highly generic and therefore widely applicable. SBST has been applied to a wide variety of testing goals including structural [Ton04, HM09], functional [WB04], non-functional [WG98] and state-based properties [DHHG06], although the most widely studied area is test generation for structural coverage [HJZ15]. Testing goals can also be combined [RCV+15, PKT18b] so that the fitness function takes into account the different properties of the class under test.

The actual search could be performed using any of the existing meta-heuristic optimization algorithms: local search algorithms (e.g., hill climbing), simulated annealing, evolutionary algorithms (e.g., GA), etc [McM04]. However, much of the literature has tended to focus on evolutionary algorithms [Har11, Ton04].

A lot of research has been conducted in the field of search based test generation and many approaches and techniques have been proposed in the last years, especially for unit testing of

object oriented programs. In the literature there are single objective (one fitness function) formulations of the problem [McM04, FA13], multi objective [HKL+10, MHJ16] formulations in which there are different opposing fitness functions to optimize and many objective formulations [PKT15, Arc17], where there is one fitness function for each uncovered test objective.

3.1.2 Web Testing Techniques

In this section we report the current state-of-the-art techniques used to generate test cases for web applications [LDD14]. In particular, we describe end to end approaches to test generation which either assume no knowledge about the web application under test (black-box approaches) or harness such knowledge to generate test cases (white-box approaches).

3.1.2.1 End to End Approaches

Model Based Approaches. These approaches create a model of the web application under test and use it for test case derivation. The model can be created manually [RT01] or it can be extracted automatically, via crawling [MvDR12, MFM13], a combination of static and dynamic analysis [MTR08] or inference from system executions [TMN+12, MTR12, TNM+13] (respectively event based inference [TMN+12] and state based inference [MTR12, TNM+13]). Paths are then extracted from the model using graph-visit algorithms that try to satisfy some model coverage criterion. Then, input values for the events in the paths are generated (or manually specified) to make those paths executable. However, both manually defined and automatically inferred models express the program semantics in an incomplete way, as a consequence of the abstraction operated during model creation [TTN14]. In fact, a complete and precise specification of the system semantics is usually unaffordable and would require a very complicated and large model, which is against the initial goal of abstracting away the implementation details to get a concise system representation. On the other hand, the approximation necessarily introduced in the model makes some of the paths derived from the model *infeasible*. Such paths violate dependencies and constraints that are not expressed explicitly in the model. Path infeasibility is one of the major open problems in model based testing [DNSVT07] and practically it is very challenging to tackle, since determining if a path is feasible is, in general, an undecidable problem.

Most model based approaches rely on *web application crawling* for the automatic construction of a navigation model and on *graph visit algorithms* (e.g. depth-first [Tar72] or breadth-first visit [Lee61]) for the selection of navigation paths that ensure high coverage of the model (e.g., transition coverage). Input data generation to turn the selected paths into executable test cases is either manual or random [TRM14]. The proposal by Mesbah et al. [MvD09] uses a crawler, Crawljax, to derive a state flow graph consisting of states and transitions that model the Ajax web application under test. States are abstraction of concrete DOM instances whereas transitions are

the (simulated) user events that lead the application from one state to the next. Then, the tool Atusa uses the inferred model to generate test cases (i.e. sequences of events). Atusa [MvD09] derives test cases from the model and makes them executable either by providing the same inputs used during crawling or by generating inputs randomly. Atusa uses the *k-shortest path* algorithm, a generalization of the shortest path problem in which several paths in increasing order of length are sought, for path selection. Specifically, Atusa collects all sink states in the crawled graph, and computes the shortest path from the index state to each of them. In this way, transition coverage in the crawled graph is achieved by construction.

Another model based approach for testing Ajax web applications has been proposed by Marchetto et. al. [MTR08]. A Finite State Machine (FSM) that models the Ajax web application is built using a combination of dynamic and static analysis. Differently from Atusa, the adopted coverage criterion, used also in GUI-testing, is based on the notion of semantically interacting event sequences. In such event sequences the ordering of events affects the state reached at the end of the interactions [YM07]. If two events interact semantically there must be some dependency between the two events. For that reason semantically interacting events are probably the most important to test and it is possible to define an adequacy criterion which is based on the coverage of pairs of semantically interacting events.

Another criterion that can be used for the generation of event sequences is sequence *diversity*. The intuition is that the more diverse the test cases are, the more effective is testing [FPCY16]. It is possible to define different diversity metrics, for instance metrics based on event execution frequencies [MT11]. A set of test cases, i.e. a test suite, can be characterized in terms of frequencies of events in each test case and the diversity of test suites (e.g. based on the average euclidean distance between pairs of vectors representing the frequencies of events in the corresponding test cases) can be optimized using search based algorithms.

Ricca et al. [RT01] generate test cases from a manually constructed *UML* model of the web application under test. The UML model is then turned into a graph and paths are extracted starting from a regular expression that represents the graph. The expression is composed of events and it can then be used to generate paths by identifying the set of linearly independent paths that comprise it, and applying heuristics to minimize the number of paths generated. A linearly independent path is defined as a path that cannot be obtained as a linear combination of the previously selected paths.

Furthermore, the problem of *feasible* paths is addressed by Tonella et al. [TTN14]. They propose to use n-grams to predict the next feasible event in a sequence in the same way as they are used to predict words in natural language processing tasks (e.g. machine translation, speech recognition, etc.). Therefore, when a sequence is built, it is possible to predict the next likely feasible n-th event to append to the sequence, based on the observation of the sequence of $n-1$ events already executed. The intuition is that, since infeasible sequences are never observed in the given corpus of real executions, the possibility of generating infeasible event sequences is reduced, though not completely eliminated, by means of n-grams.

User Session Based Approaches. In user session-based testing, testing is done by keeping track of user sessions. In particular, a list of interactions performed by a user is collected in the form of URLs and name-value pairs of different attributes, from which test cases are then generated. Since most web application operations comprise receiving requests from the client and then processing those requests, the collection of client requests can be done with minor tweaks to the server. The most significant work in this direction was proposed by Elbaum et al. [ERKI05]. They propose three different ways to use user sessions for testing purposes. First, they used session data directly, i.e. different user sessions were replayed individually. Second, they replayed a mixture of different session interactions from different users, in order to expose the error conditions caused when conflicting data is provided by different users. Finally, they proposed to mix the real user sessions with actions which are likely to be problematic (e.g., navigating backward and forward while submitting a form). Their empirical evaluation shows that such capture-and-replay techniques for user session-based testing are able to discover certain special types of faults which cannot be exposed by white-box testing.

Search Based Approaches. SBST has been applied also to web applications, e.g. by Alshahwan and Harman [AH11] to test PHP web applications. The main aim of this technique is to maximize branch coverage. The algorithm starts with a static analysis phase that collects static information to aid the subsequent search based phase. The search based phase uses an algorithm that is derived from Korel's Alternating Variable Method (AVM) [Kor90] in addition to constant seeding. The paper shows that there are many issues associated with white-box web application testing, such as dynamic type binding and user interface inference.

Feedback Directed Random Approaches. Belonging to random testing is Artemis, a framework for automated testing of JavaScript web applications proposed by Artzi et al. [ADJ$^+$11]. The framework takes into account the features of JavaScript, such as its event-driven execution model and interaction with the Document Object Model (DOM) of web pages. The framework is parameterized by: (1) an execution unit to model the browser and server, (2) an input generator to produce new input sequences, and (3) a prioritizer to guide the exploration of the application's state space. By instantiating these parameters appropriately, various forms of feedback-directed random testing [PLEB07] are proposed.

Yu et al. [YMZ15] proposed a tool called InwertGen. Differently from Artemis, InwertGen generates its test cases against the Java code of the page objects, which in turn are obtained heuristically at run time by an algorithm proposed in the same paper. Test cases are generated using the tool Randoop [PLEB07].

Symbolic and Concolic Execution Approaches. Artzi et al. [AKD$^+$08] proposed a white-box concolic testing technique for PHP web applications. Their tool is called Apollo. The aim of the technique is to find failures in HTML-generating web applications: the technique is based on dynamic test generation, using combined concrete and symbolic execution. The technique generates tests automatically, runs the tests (sequences of URLs) capturing logical constraints on inputs, and minimizes the conditions on the inputs to failing tests, so that the resulting bug reports

are small and useful in finding and fixing the underlying faults. Apollo generates test inputs for a web application, monitors the application for crashes, and validates that the output conforms to the HTML specification (it checks if the HTML page is well-formed using an HTML validator).

Considering the JavaScript programming language, Jalanji [SKBG13] is a framework which enables the dynamic analysis techniques for JavaScript programs, among which there is concolic testing. Their implementation of concolic testing supports constraints over integer, string, and object types.

While Jalanji focuses on pure JavaScript programs, SymJS [LAG14] is a framework for automatic testing of client-side JavaScript web applications. However, it uses symbolic execution, which is performed within a virtual machine, rather than concolic execution. For exhaustive testing SymJS also constructs event sequences automatically based on dynamic analysis. It contains various sequence construction schemes, one of which is a re-implementation of the main algorithm of the concrete testing tool Artemis [ADJ$^+$11].

Specification Based Approaches. Thummalapenta et al. [TLS$^+$13] present a behavioural-driven technique for generating web tests along interesting business-related behaviours of the web app. The behaviours of interest are specified in the form of business rules. Business rules are a general mechanism for describing business logic, access control, or even navigational properties of an application's GUI. The technique uses a crawler to explore the application's GUI. To handle the unbounded number of GUI states, the technique includes two phases. The first phase uses an abstraction function that represents equivalence of GUI states without considering rules. The second phase identifies rule-relevant abstract paths aiming at full coverage.

3.1.3 Limitations and Open Problems

The approaches described above present a series of limitations that pave the way to new techniques to address them.

Concolic and search based white-box approaches [AKD$^+$08, AH11] are only applicable when the web application under test is developed with the multi page paradigm, in which web pages are dynamically generated by the server side and are transmitted synchronously to the browser, where they replace the previously visited page. However, they are severely limited when the development paradigm shifts to single page, where the client handles most of the computation. Indeed, the proposed techniques were specifically designed to work with server-side scripting languages (such as PHP) and cannot be extended to work with client-side languages (e.g. Javascript). On the other hand, concolic and symbolic execution approaches targeting JavaScript programs [SKBG13, LAG14] may represent an alternative to E2E approaches and it would be interesting to compare them. However, both Jalanji [SKBG13] and SymJS [LAG14] have been evaluated on very simple JavaScript programs and web applications ($\approxeq$ 1270 LOC of non framework-based JavaScript code on average of which only $\approxeq$ 400 LOC are executable). Pros

and cons of concolic and symbolic execution approaches for web applications and the comparison with E2E approaches need to be assessed in a systematic way and we leave that for future work.

In user session based approaches [ERKI05] the level of human effort involved in collecting data is relatively small but the actual effectiveness of the technique depends on the number and on the representativeness of data collected.

Regarding feedback directed random approaches, the Artemis framework [ADJ+11] analyzes the JavaScript code of the web application under test and it has been applied to single page web applications today considered quite trivial ($\approxeq$ 700 LOC of non framework-based JavaScript code on average). The approach simulates the browser environment (using the *envjs* library, no longer maintained [1]) which may hide some of the issues a real user can experience using the real browser. Furthermore, Artemis analyzes the client side code to guide the generation of sequences and test inputs, which might be difficult if the client side is developed using frameworks/libraries. In fact, in such cases, the source code that contains the business logic of the web application can be thoroughly analyzed only taking into account the framework source code and the third party libraries that are used. Overall, even if the aforementioned limitations of Artemis could be overcome by other techniques (e.g. InwertGen [YMZ15]), the feedback directed random approaches have been shown to be ineffective for automatic generation of unit tests [SJR+15]. In particular, given the same timeout, Randoop generates a lot more test cases (i.e. two order of magnitudes more test cases) than other test generators (e.g. Evosuite [FA13], belonging to the category of search based techniques) but it achieves less coverage. The reason is that feedback-directed random approaches do not target specific test objectives. Therefore, feedback directed random approaches are not suitable for automatic generation of E2E web tests in which efficiency is crucial, since the execution of each E2E test is slower than that of, for instance, a Javascript unit test, since the execution environment of an E2E web test is the browser [2] [3].

All model based approaches share some common limitations. Most of them do not address the feasibility problem [MvD09, MTR08, MT11, RT01] which may hinder the fault detection capability of the test generator since not all test paths derived from the model can be turned into test cases that traverse the desired paths upon execution. Furthermore, existing model based approaches focus on the extraction of abstract paths from the model (graph visit algorithms [MvD09], semantically interactive events [MTR08], diversity [MT11], linear independence [RT01]) and they resort to random inputs to make those paths executable or they rely on manually specified inputs. In the latter case manual input specification is expensive to carry out for developers whereas random input generation, when it is decoupled from the abstract paths generation, may not be effective, since feasibility depends on the combination of path extracted

[1] https://github.com/thatcher/env-js

[2] https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html

[3] https://vuejsdevelopers.com/2019/04/01/vue-testing-unit-vs-e2e/

and inputs generated. To the best of our knowledge, the only approach that deals with the feasibility problem is the work proposed by Tonella et al. [TTN14]. However, feasibility is addressed only taking paths into account, without considering input values required by the events in the paths. Furthermore, the authors rely on a corpus of real executions to predict the feasibility of the next event during path generation, hence, the success of the technique depends on the representativeness of executions collected.

In conclusion, new approaches are needed to address the feasibility problem effectively and efficiently, in which the combination of paths and input values are generated jointly without relying on external inputs as a corpus of real executions or manually specified data.

3.2 Test dependency

In this section we describe works related to test dependency. In particular, we present the tools that support test dependency management and the techniques available for detecting dependencies in unit test suites developed in Java. We also show how the problem of test dependency is related to automated regression testing techniques and test flakiness. We conclude the section by motivating the need for new approaches that address dependency detection for web E2E test suites.

3.2.1 Tools Supporting Test dependency Management

Testing frameworks provide mechanisms for developers to define the context for tests. For example, JUnit, starting from version 4.11, supports executing tests in lexicographic order by test method name [Git12]. JUnit version 5 lets developers implement their own test methods order or use three built-in options: alphanumeric ordering, numeric ordering or random ordering [JUn19]. TestNG [Tes19] allows developer to specify dependencies between tests and supports a variety of execution policies that respect these dependencies. However, although such testing frameworks allow dependencies to be made explicit and respected during execution, they do not help developers identify unknown (not documented) dependencies.

Test dependencies can be caused by poorly isolated test executions. On this topic, Muşlu et al. [MSW11] identified tests that fail when executed in complete isolation. This indicates that the tests require a specific state to be in place before their execution. However, Muşlu et al. do not extract concrete dependencies. On the same line, Bell and Kaiser [BK14] present VMVM, a tool that runs multiple tests in the same JVM but selectively resets state regions that may have been written by tests such that each test runs from the initial state as if run in a separate JVM. VMVM instruments all classes and re-initializes the static fields that can be shared across tests. The goal is to speed up testing compared to running each test in a separate JVM but also to

prevent data flowing between tests via internal resources by design. Therefore, the approach effectively masks the effects of poorly designed tests (i.e. with dependencies) although, in turn, it makes the job of developers who must identify and fix such poorly designed tests harder. In fact, VMVM can avoid test dependencies by providing support for automatically resetting the state, but it does not determine if tests are actually dependent.

Dependency Detection Techniques. Different techniques have been proposed recently to detect dependencies in unit tests. Zhang et al. [ZJW+14] empirically studied test-order dependency and proposed a technique to find dependent tests in the existing test suites. Their study of issue-tracking systems for five projects found 96 dependent tests, of which 61% are due to shared state. Their tool DTDetector explores selected permutations of test suites to manifest dependent tests in JUnit test suites. Guided by the analysis of real-world dependent test suites on open source projects they developed a dependency-aware k-bounded algorithm and they found that a small value of k (e.g., $k = 1$ and $k = 2$) finds most dependent tests on real-world projects while keeping the runtime limited.

Test dependencies can also be caused by external resources, such as files and sockets that are shared between the tests. Identifying how tests interact with those external resources might give developers a better view of what causes tests to behave differently in different executions. Gyori et al. [GSHM15] proposed PolDet, which uses dynamically identified *state polluting tests*, i.e., tests whose execution results in persistent changes to the testing environment and that might influence the behaviour of other tests. PolDet focuses on finding the tests which might introduce dependencies by leaking data, but does not consider the tests which are actually affected by that. In other words,PolDet detects potential data sharing between tests, i.e. data left behind by a test that a later test may read, which may or may not ever occur.

ElectricTest [BKMD15] utilizes dynamic data-flow analysis to identify all conflicting write and read operations over static Java objects in a given test suite. Rather than detecting *manifest dependencies* (i.e., a dependency that changes the outcome of a test case [ZJW+14]), ElectricTest detects simple *data dependencies* and anti-dependencies (i.e., respectively read-after-write and write-after-read). Since not all data dependencies will result in manifest dependencies, ElectricTest is inherently less precise (over-approximation) than DTDetector at reporting *true* dependencies between tests (i.e. it could report false positives). Furthermore, ElectricTest uses dependency information (once dependencies are detected) to soundly parallelize the execution of tests.

Gambi et al. [GBZ18] propose Pradet that shares the basic approach to detect data dependencies with ElectricTest by executing the tests once and observing conflicting accesses to objects that can be reached by the tests through static references. Two main differences separate Pradet and ElectricTest. Compared to ElectricTest, Pradet precisely handles dependencies which involve String objects and enumerations, but it does not handle external data dependencies. Furthermore, differently from ElectricTest, Pradet iteratively refines data dependency to establish if they result in manifest dependencies. In fact, if tests are data dependent there is the possibility of observing

unexpected behaviours, i.e., the manifest dependencies, when they are executed out-of-order. The iterative refinement process starts out by selecting a data dependency to check. Then, it proceeds with scheduling an execution of tests such that all the dependencies, except the selected one, are respected. Next, it checks if the tests produce the same outcome although executed out-of-order. If the outcome of the tests does not change with respect to the one obtained by executing the tests in the expected order, then the data dependency is not a manifest dependency and it is discarded. Otherwise it is stored as manifest and not removed. The process is repeated until all data dependencies are removed or become manifest.

3.2.2 Regression Testing Techniques Assuming independence

Regression testing is the process of validating modified software to provide confidence that the changed parts of the software behave as intended and that the unchanged parts of the software have not been adversely affected by the modifications [HJL+01]. Rerunning all test cases in a test suite every time the software changes or there is a new release may, however, be expensive. An improvement is to apply a *test selection* technique to select an appropriate subset of the test suite to be run, for example by selecting only tests that have been impacted by changes in the application code. If the subset is small enough, significant savings in time are achieved. Dependencies in the given test suite can break existing test selection techniques since a test cannot be selected and executed alone if it depends on another.

Due to imprecision in the analysis carried out to estimate impact, test selection may be unsafe in practice and some may turn to *test prioritization*, where the entire test suite is ranked and tests most likely to be impacted by recent changes are ranked first. In fact test prioritization techniques [RUCH01] schedule test cases so that those with the highest priority, according to some criterion, are executed earlier in the regression testing process than lower priority test cases. For example, testers might schedule test cases in an order that achieves code coverage at the fastest possible rate, exercises features in order of expected frequency of use, or increases the likelihood that faults will be detected early in testing. Basically test prioritization techniques consider the set of all permutations of a test suite and find the best reward value for an objective function over that set. However, reordering test cases in a test suite assume test independence.

An approach to speed up test execution, which is complementary to test case selection and prioritization, is *test parallelization* [Kap16, BKMD15]. Test parallelization runs tests concurrently utilizing different computational resources (threads, processes, servers, etc.). Tests are scheduled in order to utilize all the available computing resources. Once again, if tests are dependent, test dependencies must be factored in the scheduling process to achieve sound results.

Test minimization, also known as test reduction, techniques aim at eliminating redundant test cases from test suites to reduce their maintenance and regression testing costs [VSM18, RHVRH02]. Most reduction techniques use adequacy criteria, such as code coverage, as a guideline to remove

whole test cases with redundant coverage [YH12]. However, removing test cases is not safe when tests are dependent as the remaining tests may break because of missing dependencies.

In conclusion, regression testing techniques that rely on the test independence assumption need to be reformulated, taking into account, in the optimization process, the dependencies between tests. For example Haidry and Miller proposed a set of test prioritization techniques that consider test dependency [M+12]; a dependency-aware test parallelization technique was proposed by Bell et al. [BKMD15].

3.2.3 Test Flakiness

Test dependencies are one root cause of the general problem of flaky tests, a term that refers to tests whose outcome is non-deterministic despite the software under test being so [LHEM14, MC13]. Recent studies conducted at Microsoft [HN15] and at Google [MGN+17] have shown that flaky tests exist in practice and lead to many broken builds. Luo et al. [LHEM14] investigated possible root-causes of flaky tests and defined a taxonomy of ten common root-causes. They found that test order dependency is one of the top three common causes of flakiness (specifically 12% of the analyzed flakiness reports). In a more recent study, Palomba and Zaidman [PZ17] showed that test smells correlate with flaky tests and can help locating them. The authors identified a strong correlations between test order dependency issues and the indirect testing smell [vDMBK02].

3.2.4 Limitations and Open Problems

All the approaches described above address the problem of test dependency for Java software. In particular the dependency detection techniques are tailored on the peculiarities of Java programs. For example DTDetector [ZJW+14] executes the test cases and records the reads and writes on global objects shared between tests. Similarly, ElectricTest [BKMD15] and Pradet [GBZ18] utilizes dynamic data-flow analysis to identify all conflicting write and read operations over static Java fields. On the contrary, in E2E web tests, the semantics of read and write operations is implicit and mediated by multiple layers of indirection such as client-side DOM, server-side application state, database entries, and remote service calls. Furthermore, with web E2E test suites, applying thorough data-flow analysis is neither feasible nor straightforward, due to the heterogeneity of technologies and languages used in modern web applications.

In conclusion, new approaches are needed to address the problem of dependency detection for web E2E test suites since state-of-the-art dependency detection techniques for Java programs, based on static analysis, are not directly applicable.

Chapter 4

Test Case Generation

The main goal of E2E test case generation when the program under test is a web application is to ensure that the functionalities of the web application are fully exercised by the generated test cases. Usually no explicit navigation graph is available to guide the creation of test cases and to measure the degree of navigation coverage achieved. Existing approaches resort to web crawling in order to build the missing navigation model [MvDR12]. However, crawling is severely limited in its ability to fully explore the navigation graph, which depends on the input generation strategy. Such strategy is usually manual input definition, random input generation, or a mixture of the two. Another limitation of crawling based approaches is that not all paths in the crawled model are *feasible* (i.e., admit a test input that traverses them upon execution) (see Section 2.3.1 for the definition of feasibility). As a consequence not all test paths derived from the crawled model can be turned into test cases that traverse the desired paths upon execution.

Contribution. In this chapter we address the problem of navigation graph construction by taking advantage of the Page Object (PO for short) design pattern (see Section 2.2.1.3), widely used in web testing [1]. The PO pattern was introduced to increase the *maintainability*, *reusability* and *readability* of test code [LCRT16]. However, it gives another indirect benefit: when defining the POs for a web application, developers implicitly define also its navigation structure, since navigation methods in POs return the next PO encountered after triggering the navigation action. We resort to such property of POs to build the navigation graph to be covered by the automatically generated test cases.

Moreover, differently from crawling based approaches [MvDR12], which first extract paths from the graph and then randomly generate the inputs for those events in the extracted paths, we perform path selection and input generation at the same time. Such joint generation of paths and inputs is guided either by a fitness function, designed to maximize the transition coverage of the navigation model, or by the diversity among the executed test cases. The guidance towards *good*

[1] `https://github.com/SeleniumHQ/selenium/wiki/PageObjects`

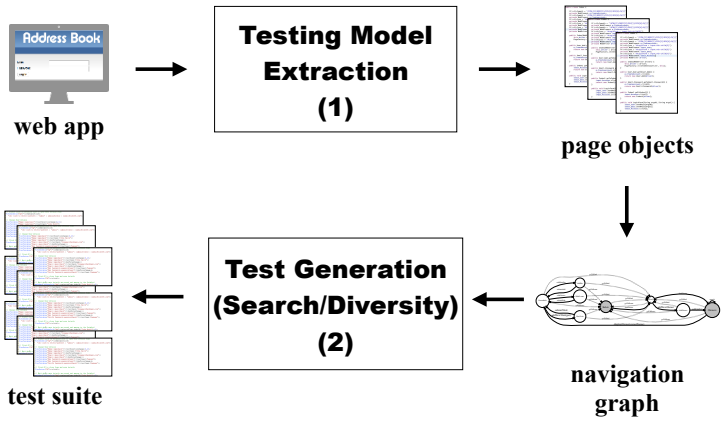


Figure 4.1: E2E web test generation approach

test cases (either through fitness or diversity) is crucial for increasing test effectiveness [FA13, FPCY16] (e.g. in terms of coverage) with respect to random approaches (as crawling based approaches). Therefore, it is also beneficial to the feasibility of the paths that are generated.

4.1 Overall Approach

In a nutshell, our goal is to effectively generate E2E web test cases that exercise application behaviours adequately. Figure 4.1 depicts the steps of our test generation approach. First, a testing model of the WAUT is extracted in the form of POs ❶. Then, from the POs we extract the model of the WAUT (i.e. the navigation graph) that is the input of the test generation component which outputs the generated test suite ❷.

Crawling based approaches, instead, generate the navigation graph by crawling the WAUT. The crawler, indeed, is able to expose the visible and hidden portions of the WAUT, as well as, the connections among web pages. However, the testing model created by the crawler presents two issues.

The first issue regards the *size* of the navigation graph that can be huge. The default state abstraction function of the crawler (see Section 2.3.2.1), which compares the equality of the string representation of the DOM of each web page, is affected by minor GUI changes leading to the presence of many DOM states (web pages), conceptually clones of each other. Basically, when the crawler visits the same page with different input data, the crawler often creates different DOM states, even though the page is conceptually the same [SLRT17, RT01, Ric04, TRM14] (this is because the state abstraction function returns 0.0, i.e. *the two states are equal*, only if the two DOM strings are exactly the same; 1.0, i.e. *the two states are different*, otherwise. The threshold in this case is not meaningful, since string equality returns a boolean value). The number of tran-

sitions, each one representing an event on a candidate clickable element, can correspondingly be potentially very large, since a new transition is created and added to the graph to connect the state before the event occurs and the state after the event occurred. Therefore, the inaccuracy of the state abstraction function creates *redundancy* that can hinder the effectiveness of the subsequent test generation step, which is based on the testing model. The risk is that of exploring the same parts of the WAUT without exercising it thoroughly.

The second issue is *state dependencies* which cause infeasibility. The crawler explores the WAUT without resetting its state and, at the end of the exploration, it outputs the navigation graph of the WAUT. Then, the test generation phase is activated, which starts from a clean state of the WAUT. Paths are extracted from the navigation graph (see Section 2.3.1 for the definition of path) but, some of them, cannot reproduce the state of the WAUT created during crawling since the state of the WAUT during test generation is different (i.e. such paths are infeasibile). For instance, let us consider the running example web application, *Phoenix Trello* (see Figure 2.9) in Figure 4.2. Let us say that the first navigation of the crawler, among other actions, logs into the web application and creates a board (respectively actions ❶ and ❷ in Figure 4.2). Then, the crawler goes back to the initial login page (top screenshot in Figure 4.2) and a new navigation starts. Let us say that the crawler signs in as a different user (respectively actions ❸ and ❹) and finds a boards page which is different from the empty boards page encountered in the first navigation (screenshot named BOARDSPAGE). Let us suppose that the crawler created a new state in the graph during the first navigation for the boards page with one board in it (named BOARDSPAGE instead of EMPTYBOARDSPAGE), clicks on it (action ❺) and the navigation proceeds. After crawling, let us suppose that the test generator extracts the test path in which the user signs in and clicks on the board in the non-empty boards page (i.e. the path composed of the actions ❸–❹–❺). Such path is not feasible during test generation since the boards page is empty. In such situation, when the test tries to click on the board which is expected to be in the web page, the Selenium driver throws an element not found exception.

By using the PO pattern, instead, all the DOM states that logically represent the same web page are grouped together in the same PO. As a consequence, each PO contains only the methods for the web page (or portion of web page) that the PO is modelling. Hence, by extracting the navigation graph from the POs, the redundancy is reduced (or even eliminated). For instance, the boards page in the *Phoenix Trello* web application is represented by the same PO both when it is empty and when it contains at least one board. In such case the method that clicks on a board b in the boards page, leading to another PO, can be executed without exceptions only if the boards page contains the board b. Therefore, the transition in the graph represented by such action can be considered as traversed/covered only if such *precondition* is respected. However the PO based model does not solve the path infeasibility problem since the model abstracts away parts of the program semantics to favour a compact and concise system representation. Therefore, the model does not represent all the constraints and dependencies of the web application under test. We tackled the path infeasibility problem from two different points of view, namely search based and diversity based approaches.

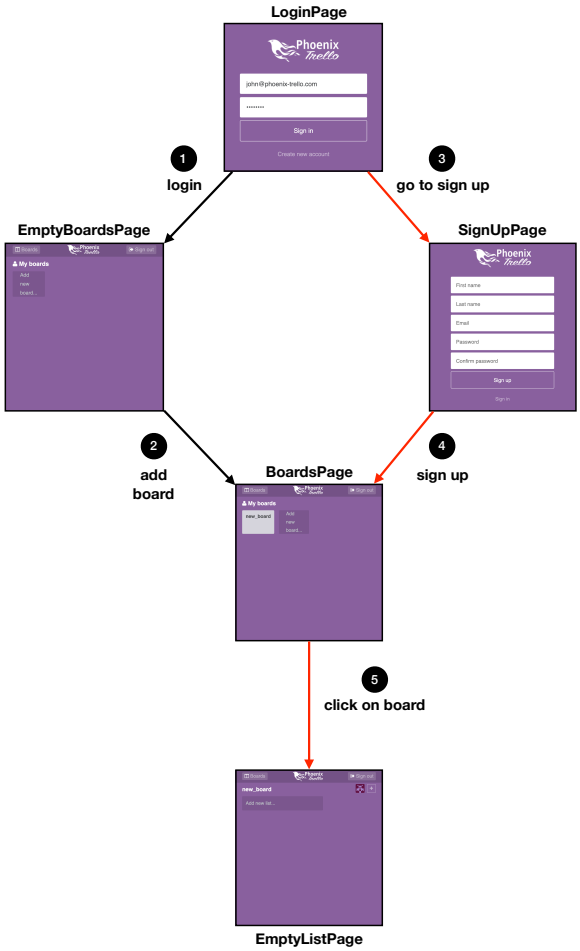


Figure 4.2: Example web crawling applied on *Phoenix Trello*. The path with red arrows, whose states are created during the second navigation of the web crawler, is not feasible during test case generation.

4.2 Testing Model Extraction

The first step of our approach is the extraction of the testing model in the form of POs. This step can be performed manually by testers, or automatically. Previous research [SLRT17] shows that POs can be created automatically with a good degree of accuracy. In order to extract the POs from a given WAUT, the tool Apogen [SLRT17] uses a crawler (i.e. Crawljax [MvDR12]) to reverse-engineering the navigation graph of the WAUT. Then, similar web pages are clustered into syntactically and semantically meaningful groups. The crawled navigation graph and the additional information, e.g. DOMs and clusters, are statically analyzed to generate Java POs, via model to text transformation.

```
1  public class BoardsPage implements PageObject {
2
3    public WebDriver driver;
4    public BoardsPage(WebDriver driver) {
5      if(!this.isPageLoaded()){
6        throw new IllegalStateException("BoardsPage not loaded ↩
             properly");
7      }
8    }
9
10   public BoardListPage clickOnBoard(String boardName) {
11     this.driver.findElement(By.id(boardName)).click();
12     return new BoardListPage(this.driver);
13   }
14
15   @Override
16   public boolean isPageLoaded() {...}
17 }
```

Figure 4.3: BoardsPage PO

Whether created manually or automatically, POs implicitly specify a navigation graph for the WAUT. In fact, one of the best practices recommended for PO creation requires that PO navigation methods return the PO of the next page upon invocation [vD15]. This means that POs specify the navigation structure of the web application under test in terms of method invocations (i.e., operations executed within each abstract web page) and page objects returned by the invoked methods (i.e., next PO reached during navigation in the web application). We use such implicit navigation graph for automated test case generation.

The API of a PO is application-specific and provides an abstraction of the concrete HTML page functionalities to the test case. Despite the term "page" object, these objects are not necessarily built for an entire page. In fact, a PO may wrap an entire HTML page or a cohesive fragment that performs a specific functionality. The rule of thumb is to group and model the functionalities offered by a page as they are perceived by the user of the web application. Furthermore, the PO developer also decides which functionalities to model that are worth testing.

Let us consider our running example web application *Phoenix Trello*. Figure 4.3 shows a portion of the code of the PO `BoardsPage`, which models the page that lists the boards owned by a user (last screenshot on the bottom right corner of Figure 2.9).

Among others, this PO contains the method `clickOnBoard` that models the user action consisting of the selection of a specific board from the board list displayed in the boards page. The actual selection is performed at line 11 where Selenium WebDriver's APIs are used to locate and operate some web elements inside the concrete HTML page of the web application. Specifically,

the web element of interest is located by its unique identifier, by means of the Selenium method `findElement(By.id(...))`. The action performed on the web element located by id is a click (Selenium method `click`, still at line 11). Since after the click the navigation continues on the next page, which is modelled by the PO `BoardListPage`, method `clickOnBoard` returns a new instance of the PO reached after the click, of type `BoardListPage`.

In general, PO methods may return values of any type (*void, int, String*, etc.). However, a recommended best practice is that *navigational* PO methods return the next PO encountered in the navigation (`this` if navigation does not leave the current PO). In the following, we call a *navigational method* any PO method that returns a PO.

The interface `PageObject`, which all POs have to implement, has the method `isPageLoaded` which has to be overridden by all POs. The aim of such method is to make sure that the actual DOM page which is loaded during the navigation is represented by the desired PO. This way we can easily check if the navigational methods are implemented correctly since the method `isPageLoaded` is called when the PO is instantiated; if the DOM page is not the desired one, an exception is thrown. In order to do that the PO developer has to identify a particular element (or a set of elements) that uniquely identify the DOM page represented by the PO. For example, the `BoardsPage` PO can be uniquely identified by the DOM element which is needed to add a new board (see last screenshot of Figure 2.9, in particular the DOM element with text *Add new board...*).

The `PageObject` interface is also useful to parametrize the execution of a PO method. For example, the `signIn` of the PO `LoginPage` in Figure 2.13 can be parametrized to include both the scenario of the successful login (i.e. credentials are correct) and the scenario of the unsuccessful one (i.e. credentials are not correct). The return type of the `signIn` method is the interface `PageObject`: in the scenario of the unsuccessful login, distinguished by checking if the new desired page is loaded (in this example, `BoardsPage`), the returned PO is `LoginPage`; otherwise the returned PO is `BoardsPage`.

Intuitively, the navigation graph is obtained from the POs by associating nodes to page objects and edges to navigational methods. More specifically, given a navigational method that, starting from a PO node, leads either to the same PO or to another PO, such method induces either a self loop edge or an edge to another node (corresponding to the returned PO) in the graph. Formally, we can define the navigation graph and its relation with POs as follows:

Definition 1 (Navigation Graph) *A navigation graph $\mathcal{G}$ for a web application $\mathcal{W}$ is a labelled, directed graph, denoted by a 4-tuple $\langle r, \mathcal{N}, \mathcal{E}, \mathcal{R} \rangle$ where:*

1. *r is the root node (called Index) representing the initial DOM state when $\mathcal{W}$ has been fully loaded into the browser.*
2. *$\mathcal{N}$ is a set of vertices representing the nodes. Each $n \in \mathcal{N}$ represents an abstract DOM state of $\mathcal{W}$.*

3. *$\mathcal{A}$ is a set of directed edges between vertices, which we call actions. Each $(n_1, n_2)_{[r]e} \in \mathcal{E}$ represents a possible transition between two nodes n_1, n_2 if and only if node n_2 is reached by executing the action e in node n_1 and the guard r is satisfied.*

4. *$\mathcal{R}$ is a set of guards on actions $(n_1, n_2)_{[r]e} \in \mathcal{E}$.*

5. *$\mathcal{G}$ can have multi-edges and be cyclic.*

Given a set of page objects P_O, the nodes $\mathcal{N}$ of the navigation graph $\mathcal{G}$ are bijectively mapped to P_O by the function $po : \mathcal{N} \to P_O$. Each edge $e \in \mathcal{A}$ connects a pair of nodes (n_1, n_2) such that the page object $po(n_1)$ contains a `return` *statement whose returned type is $po(n_2)$.*

Algorithm 1: Navigation graph extraction algorithm

```
1  Procedure extractNavGraph(G, po)
       Input:
       G: navigation graph computed so far
       p_o: page object to be analyzed
       Output:
       G: updated navigation graph
2      begin
3          n_1 := getNodeByPO(G, p_o)
4          l := getNextPOsByStaticAnalysis(p_o)
5          v = ∅
6          for p'_o ∈ l do
7              n_2 := getNodeByPO(G, p'_o)
8              if n_2 = NULL then
9                  n_2 := newNode(p'_o)
10                 G.N := G.N ∪ {n_2}
11                 v := v ∪ {n_2}
12             end
13             G.E := G.E ∪ {⟨n_1, n_2⟩}
14         end
15         for n_2 ∈ v do
16             extractNavGraph(G, mapNodeToPO(n_2))
17         end
18     end
```

Algorithm 1 shows the recursive navigation graph extraction procedure. The loop at lines 6–12 iterates over all POs that are possibly returned by the PO under analysis. The set of such POs

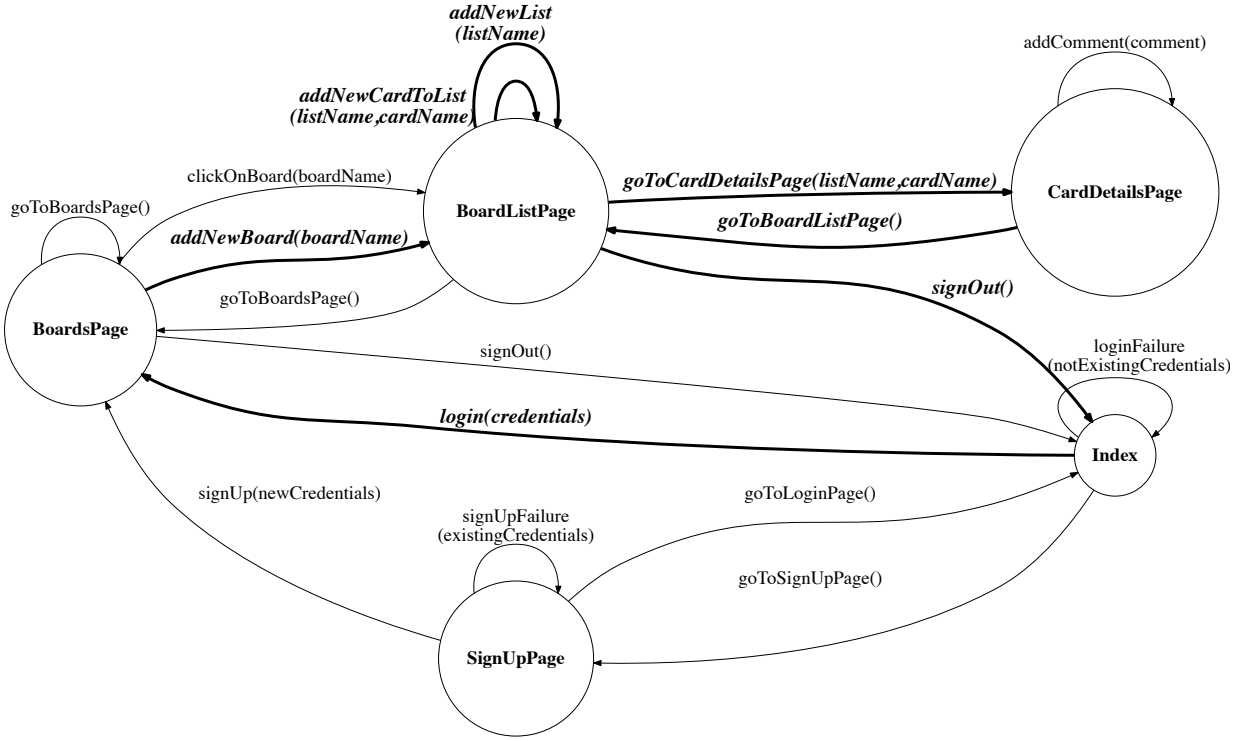


Figure 4.4: Simplified navigation graph of the *Phoenix Trello* web application. Transitions between states are labelled with the corresponding events, which can be parameterized. The path in bold represents an instance of transition coverage-maximizing test path, which are those our test generation techniques aim to generate. For simplicity we do not show the guards in the graph.

is obtained by static code analysis (line 4). When the returned PO is not already mapped to a graph node, a new node is created (line 9) and added to the graph (line 10). An edge (i.e. action) $\langle n_1, n_2 \rangle$ from the node n_1 associated with the PO under analysis to the returned PO node n_2 is then added to the graph (line 12). Graph extraction continues recursively on all newly created PO nodes (stored in variable v), i.e., all PO nodes not already present in the initial graph $\mathcal{G}$ (lines 13–14). Note that the procedure `getNextPOsByStaticAnalysis` can also navigate the PO method in order to handle the case of a parametrized PO method. In particular, if a PO method has the returned type `PageObject`, the PO method is traversed to collect all the returned types (which are POs) of the method.

Figure 4.4 shows a simplified navigation graph of the *Phoenix Trello* web application. The graph consists of five abstract DOM states, each having several possible actions that can trigger transitions between them. Since the navigation graph is obtained by statically analyzing the POs, each action is a PO method.

Next, we formally define the notion of feasibility, and its impact on automated test case generation.

4.2.1 Guards

Given an abstract DOM state n, the set of possible actions $\mathcal{E}$ may be constrained by one or more guards (*a.k.a.*, preconditions).

Definition 2 (Guard) *A guard $r(n, e, x)$ is a boolean condition over the possible input values $x \in X$ of an action e in a specific application state n.*

The guard needs to be satisfied in order to enable the transition between state n and the target state of the action e (typically another state n', or n itself). If the guard r of the action e is not satisfied, the action e throws an exception/error. The application state n includes the global variable values, the DOM, and any persistent data that remains available across user interactions.

Hence, satisfiability of a guard depends on: (1) the input values $x \in X$ generated for the given action e, as well as, (2) the internal business logic state n of the application, produced by previous user interaction sequences.

For instance, in the `BoardsPage` state of Figure 4.4, an action `clickOnBoard` allows to navigate towards the `BoardListPage` state. A simple guard over this action imposes that the board identified by the input `boardName` must be present in the application prior to executing the event. (For readability, we omitted the guards in the figure.) As such, the guard depends not only on the specific value assigned to the input `boardName` of the `clickOnBoard` action, but also on the internal business logic state of the application. In fact, the board to be clicked must have been previously inserted into the application by a dedicated action (e.g., by executing the `addNewBoard` action in the same `BoardsPage` state).

4.3 Test Generation Problem Definition

Given a navigation graph of a web application, the goal of a web test generator is to automatically extract sequences of actions and generate suitable inputs in order to exercise the application behaviours thoroughly, potentially covering all transitions in the navigation graph.

Definition 3 (Test Path) *A Test Path is a sequence of test states paired with a corresponding sequence of actions with unspecified input values.*

Then, a test case can be defined by instantiating concrete input values within a test path.

Definition 4 (Test Case) *A Test Case is a sequence of concrete values for a corresponding sequence of actions in a specific test path.*

For example, in *Phoenix Trello*, a simple test path that displays a board for a certain user consists of the test state sequence ⟨`Index`, `BoardsPage`, `BoardListPage`⟩. The corresponding action sequence is given by ⟨`login`, `clickOnBoard`⟩.

In such sequence of actions, two input values need to be specified: (1) `credentials`, the credentials of the user to insert (namely username and password of an already registered user), required by the action `login` of state `Index`, and (2) `boardName`, the name of the new board to add, required by the action `clickOnBoard` of state `BoardsPage`. The action `clickOnBoard` has a guard (not shown in the graph) that states that the board identified by `boardName` must exist before the execution of the action. Since in the considered test path no board is added by any event before executing `clickOnBoard`, the chosen test path cannot be taken, for any possible values of input `boardName`. Therefore, we say that such path is *infeasible*.

Definition 5 (Test Path Feasibility) *A Test Path is feasible if there exists an input parameter-value assignment that satisfies all the guards related to the actions in the test path. If such input parameter-value assignment satisfying all the guards in the test path does not exist for every possible input value, then we say that the test path is infeasible. A concrete test case derived from an infeasible test path, upon execution diverges (i.e. do not traverse/cover), from the path it is supposed to traverse.*

In conclusion, the test generation problem that we address (for the transition coverage adequacy criterion) is then to generate a set of feasible test paths, as well as the related parameter-value assignment, which, upon execution, ensure that all navigation graph edges are traversed at least once. In this thesis, we found that feasible test paths are produced by means of search based and diversity based approaches.

4.4 Search Based Web Test Generation

The first concern with regards to search based software testing is the definition of a fitness function that captures the test objective, namely transition coverage. Such fitness function is used to guide a search based optimization algorithm which searches the space of test inputs (test paths in conjunction with the corresponding input values) to find those that maximize transition coverage. In particular we used a genetic algorithm [Ton04, FA13, PKT18a] which performs test paths selection and input generation at the same time.

The fitness function establishes the *quality* of each test case. In order to have a test case that maximizes transition coverage such test case has to be feasible. Therefore, the fitness function can be defined as a metric that quantifies the degree of feasibility of a test case and guides the search towards feasible test cases.

```
1  public class BoardsPage implements PageObject {
2
3     public WebDriver driver;
4     public BoardsPage(WebDriver driver) {...}
5     public List<String> getExistingBoardNames() {...}
6
7     public BoardListPage clickOnBoard(String boardName) {
8        if(getExistingBoardNames().contains(boardName)) {
9          this.driver.findElement(By.id(boardName)).click();
10         return new BoardListPage(this.driver);
11       } else throw new IllegalArgumentException();
12    }
13    ...
14 }
```

Figure 4.5: BoardsPage PO with guard in the goToBoard method

4.4.1 Guards Specification in PO Methods

According to the definition of feasibility (see Definition 5), a test path is feasible if all the guards related to the actions in the test path are satisfied by a generated parameter-value assignment. One way to guide the generation of feasible test paths is to explicitly specify the guards in the actions of the navigation graph, hence in the PO methods.

Specifically, each navigational method in each PO should specify the condition (i.e. the guard) under which it can be safely executed. Such condition may depend on the invocation parameter values, as well as the state of the application, which is determined by the actions performed on the application in the previous navigation steps. In Figure 4.5, the guard of method `clickOnBoard` deals with the proper selection of a board from the list of boards shown in the boards page. Each board is uniquely identified by its name `boardName`. In the running example, the valid value for the `boardName` parameter must be in the list of already created boards, whose names are returned by the method `getExistingBoardNames`. If the guard is not respected, an exception is thrown.

Guards depend on the WAUT business logic and intended behaviour and thus cannot be generated fully automatically. While the requirement that every navigational method returns the next PO is a best practice which is commonly followed (although it is not enforced by the PO pattern), the inclusion of guards is more impactful. Indeed, in practice, developers write test code that respect guards by construction, making them not strictly necessary. However, guards are a good programming practice, independently of the use of our technique.

In conclusion, in order for a test path $P = \langle S, A \rangle$ (see Section 2.3.1 for the definition of test path) to be feasible, the conjunction of the constraints in the method guards associated with the

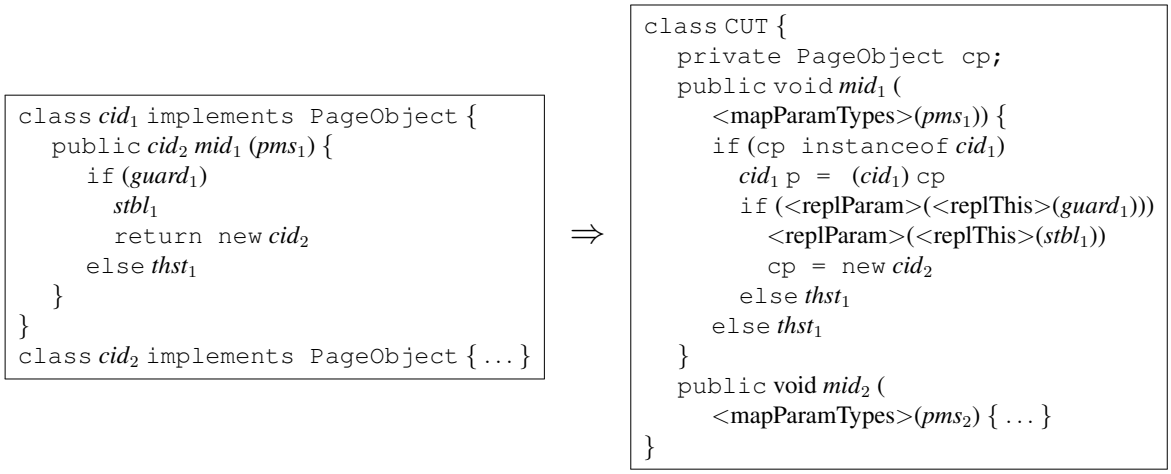


Figure 4.6: Automated program transformation that generates class `CUT` from the POs

edge sequence A must be satisfiable. Since some of the values evaluated in the method guards may depend on the server/client side state (e.g., `getExistingBoardNames()` in Figure 1), in general the problem of determining whether a test path P is feasible or not is an undecidable problem. Moreover, since feasibility depends on the server/client state, which is computed by arbitrarily complex programs, SAT solvers are generally not a viable tool to address the test path feasibility problem. For these reasons, we resort to a meta-heuristic algorithm.

4.4.2 Problem Reformulation

The problem of generating test cases that cover all navigation graph edges can be reformulated as a standard branch coverage problem on an *artificial class* generated from the navigation graph and the POs. In fact, a path $P = \langle S, A \rangle$ consists of a method sequence (namely, the sequence of method invocations associated with A), for which suitable parameter values X must be found. Hence, we can solve the feasible path generation problem and the parameter input value generation problem by applying the search based approaches that have been proposed for object oriented testing [FA13], where method sequence and parameter values are generated at the same time. This requires the creation of an artificial class under test (which we call `CUT` for short) whose methods are the methods associated with the navigation graph edges and whose state is the currently visited web page and more specifically, the currently instantiated PO for such web page.

Figure 4.6 shows the program transformation that creates class `CUT`. Its input is a set of POs and its output is class `CUT`, containing a private field to store the current page object, `cp`. Each PO method becomes a method of the new class, whose return type becomes `void`. The method can be called only if the current PO `cp` is an instance of the PO where the method originally

```
1  public class CUT {
2    private PageObject currentPage;
3
4    public void clickOnBoard(BoardName boardName) {
5      if(this.currentPage instanceof BoardsPage) {
6        BoardsPage page = (BoardsPage) this.currentPage;
7        if(page.getBoardNames().contains(boardName.value)) {
8          page.driver.findElement(By.id(boardName.value)).click();
9          this.currentPage = new BoardListPage(this.driver);
10       } else {
11         throw new IllegalArgumentException();
12       }
13     } else {
14       throw new IllegalStateException();
15     }
16   }
17
18 }
```

Figure 4.7: Excerpt of `CUT` generated from the POs of the Phoenix Trello example

belonged to. When this condition is satisfied, the current page object is cast to its concrete type and assigned to the local variable `p`. This variable must replace any occurrence of `this` in the body of the original method, including its guard $guard_1$. This is performed by function <*replThis*>. The instruction that returns a new PO in the original code is transformed into a statement that assigns such new PO to the class field `cp` (*current page*) of `CUT`.

To facilitate the job of the test generator, the original parameter types (e.g., `boardName:String`) are mapped to a type with smaller range (e.g., `boardName:BoardName`, a custom type that contains a user specified number of string values) by function <mapParamTypes>. Such smaller range can be determined by static analysis, in simple cases in which the range is specified in the guard (e.g. $id \geq 1 \wedge id \leq 6$), or it can be specified by the tester (e.g. the `BoardName` type). As a consequence, any occurrence of the original parameter identifiers must be replaced with an accessor to the parameter value (e.g., `x` becomes `x.value`). This is performed by function <replParam>. Figure 4.7 shows the result of the transformation when it is applied to the PO in Figure 4.5.

We apply search based test case generation as instantiated for object oriented systems [FA13] in order to find the method sequences and parameter values that cover the last statements of the transformed method bodies, which correspond to the statements returning a new PO in the original methods (i.e., `this.currentPage = new BoardListPage(page.driver)` for `clickOnBoard`). In fact, coverage of all the statements that return the next PO in the navigation is equivalent to covering all the edges in the navigation graph, i.e., to transition coverage. In particular, we use a Genetic Algorithm (GA) and the evolved chromosomes are test suites,

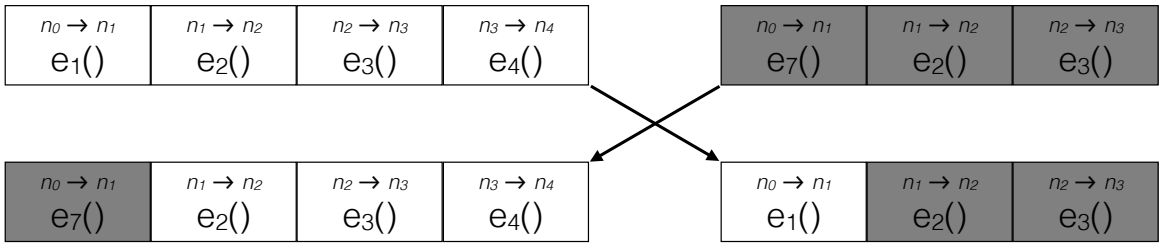


Figure 4.8: Crossover operator

i.e., set of test cases which are sequences of method calls. The fitness function is the sum of the branch distances of the yet uncovered branches [FA13]. On the other hand, the standard genetic operators for object oriented test generation do not work properly in our case, because they do not take the structure of the navigation graph into account. Hence, we have defined new crossover and mutation operators, described in the next section. The initial population is obtained by performing multiple random walks on the navigation graph.

4.4.3 Genetic Operators

We defined new genetic operators with the aim of modifying the test cases during evolution, taking into account the constraints imposed by the navigation graph.

Crossover. We have defined a crossover operator that works at test case level, in addition to the usual test suite crossover operator [FA13]. Our new crossover operator is shown in Figure 4.8, where the notation $n_i \rightarrow n_j$ above the method name $e_k()$ indicates that method $e_k()$ has PO n_i as starting node and PO n_j as target node. Crossover is straightforward to apply if the cut point selected on the two tests is between method calls that refer to the same PO (in Figure 4.8, the cut point between $e_1()$ and $e_2()$ in both tests refer to the same PO, n_1, which is the target of $e_1()$ and the source of $e_2()$). When this does not happen, the two different POs are connected by performing a random walk in the hammock subgraph between them. To ensure reachability during the random walk, head and tail of the new test are possibly shortened, until reachability holds between the two POs.

Mutation. We have maintained the test suite mutation operator [FA13], but we have modified the delete and insert method call operators, which work at the test case level. An example of how they manipulate the test is provided in Figure 4.9. The *change* method call operator is applicable only if the alternative method has the same source and target POs as the original method call.

The *delete mutation operator* randomly selects a starting method from the test case and, given the target PO of the selected method (in Figure 4.9, method $e_2()$ and target node n_2), it removes

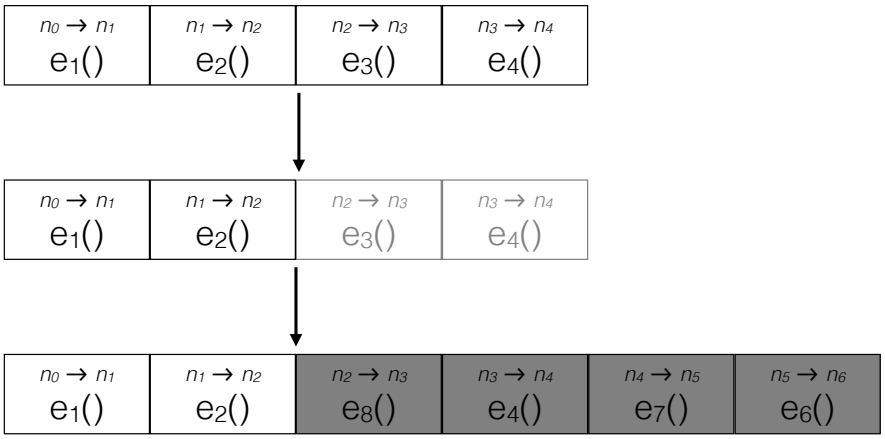


Figure 4.9: Mutation operator: *delete* followed by *insert*

all the following method calls until it finds one with a source node that is equal to the target node of the starting method. If it does not find it, it deletes all the methods from the selected point until the end of the test (as in Figure 4.9). This operation cannot remove all the statements (at least one, the first method call, is always left), to avoid the generation of an empty test case.

The *insert mutation operator* always starts at the end of the test case (in Figure 4.9, method $e_2()$, which has become the last method call after application of the *delete* operator) and it selects a method corresponding to a yet uncovered branch (e.g., $e_6()$). Then it performs a random walk on the hammock subgraph between the target node and the source node of the two selected methods (i.e., the hammock subgraph between n_2 and n_5). The path obtained in such random walk is appended to the chromosome (in Figure 4.9, methods $e_8()$, $e_4()$, $e_7()$, plus the target method $e_6()$). If the source node of the uncovered method is unreachable from the end of the chromosome, the insert operator fails and does not change the chromosome.

Insert and delete operations balance each other, by extending and shrinking the test cases, hence providing a mechanism for bloat control [FA13] (*bloat* occurs when negligible improvements in the fitness value are obtained by extremely large solutions).

4.4.4 Implementation

We have implemented SUBWEB on top of Evosuite [FA13]. In particular, we have enabled the *Whole Test Suite* strategy, because we have multiple targets to satisfy. We have modified Evosuite in order to take the navigation graph into account, both when generating the initial random population of individuals and in the genetic operators, which must generate method sequences compliant with the navigation graph.

We use *Selenium WebDriver* to instantiate the driver needed to launch and send commands to the browser, when test cases have to be executed in order to measure their fitness. The constructor of the class under test contains a method that instantiates the Selenium driver and resets the state of the application (e.g., ensuring the database is initially empty).

4.5 Empirical Evaluation

The goal of the case study is to assess pros and cons of the proposed approach. The baseline for comparison is the navigation graph produced by a state of the art crawler, Crawljax [MvDL12], and the test cases derived from such graph. We have formulated the following research questions:

RQ_1 (Cost): *What is the size of the POs to be written manually and what is the size and complexity of the PO method guards, required by our approach?*

RQ_2 (Navigation Graph): *How does the navigation graph specified through POs differ from the navigation graph obtained through crawling?*

RQ_3 (Test Suite Features): *What is the size of the test suite generated by* SUBWEB *as compared to that derived from the crawled navigation graph and what is the proportion of divergent test cases?*

RQ_4 (Coverage): *What is the level of coverage reached by the test cases generated by* SUBWEB *in comparison with the coverage reached by the test cases derived from the crawled navigation graph?*

4.5.1 Subject

AddressBook [2] is a web-based address and phone book, contact manager and organizer. It is written in *PHP* and it uses *JavaScript* for handling and modifying *HTML* elements at runtime; moreover it is backed by a *MySQL* database. The web application is a multi page (non framework-based) web application with $30k$ *PHP* LOC and $1.3k$ *JavaScript* LOC. Moreover, this application has been used as a case study in previous works on web application testing [LSRT16, LSRT15].

4.5.2 Procedure and Metrics

RQ_1 (Cost). We manually created the POs for the *AddressBook* web application, since such subject is not equipped with PO-based test suites. To minimize any subjectivity/bias, we developed the POs by adopting a rigorous procedure. Specifically, we adhered to the guidelines

[2] `https://sourceforge.net/projects/php-addressbook/`

given by Van Deursen [vD15] on the design of PO-based web test suites. Each PO represents a test state, with explicit responsibilities for state navigation and state inspection. Thus, we represented each action of a test state as a *navigational method* in the PO. Instances of such methods are, for instance, clicks and data-submitting forms that bring the browser to a new state (e.g. `clickOnBoard` method in Figure 4.5). *Getter methods* have been used to retrieve the value of key/unique elements displayed in the browser when it is in a given state (e.g. `getExistingBoardNames` method in Figure 4.5). These are usually used in test scenarios to check the feasibility of a navigational method (e.g the getter method `getExistingBoardNames` is used in the guard of the navigational method `clickOnBoard` in Figure 4.5).

In light of these design considerations, we modelled the web applications into POs as follows. Starting from the root node *Index* (*i.e.*, the initial web page, typically the login page), we modelled it as a PO. Following the navigations that were possible from the initial state, new test states were discovered (e.g., for user registration), which in turn were modelled as POs. Then, we proceeded following the actions that were possible in each newly discovered state, building POs iteratively and incrementally, until all the pages were accounted for. Additionally, states having common behaviours (e.g., menu bars) were organized into reusable components [vD15].

To analyze the manual cost that a tester incurs when using our approach, we measure the lines of code (LOC[3]) of all POs needed to model the subject application. In particular, we are interested in the manual cost for writing the guards, since they represent a specific requirement of our approach. We measure the total number of guards, the total number of logical operators in such guards and the lines of code of methods used exclusively by the guards.

RQ_2 (Navigation Graph). Since the navigation graph extracted from the POs is specified directly by the testers, we assume it as the reference and we measure the difference between the crawled graph and such a reference, in terms of graph size, states/transitions missing in the crawled graph and split/merged states/transitions in the crawled graph as compared to the PO navigation graph. The purpose of this research question is to understand whether crawling alone, with no human involvement for PO definition, is able to produce a navigation graph close to the ideal one, specified through the POs.

RQ_3 (Test Suite Features). Test case derivation from the navigation graph produced by Crawljax is supported by the tool Atusa [MvD09], which is unfortunately unavailable. Hence, we have reimplemented the test derivation algorithm of Atusa by following its description in the reference paper [MvD09]. We call our reimplementation Ext-Crawljax. We are interested in comparing the size of the test suites produced by SUBWEB vs Ext-Crawljax. A smaller size is preferable because it makes manual oracle creation or validation easier for testers. Moreover, we measure the proportion of divergent test cases (i.e., those that upon execution do not cover the test path for which they were generated, hence the corresponding test paths are likely to be infeasible). In

[3]Non-commenting lines of code, calculated by *cloc* (`https://github.com/AlDanial/cloc`)

Table 4.1: RQ_1: cost of writing POs and its guards

POs			Guards		
#	LOC	# Navigational Methods	#	Method LOC	# Logical Operators
13	764	73	16	75	54

fact, the occurrence of divergences is detrimental to the actually achieved coverage, with respect to the theoretical coverage guaranteed by the test case derivation algorithm.

To measure test case divergences, we transform each path obtained from the crawled navigation graph into a JUnit test case. The JUnit test case fires a sequence of events that should bring the application from the initial to the end state of the path. If an event is a form submission, we insert all the needed input values (either random or custom values, when necessary). The execution of such test case is deemed divergent when a *Selenium* exception is thrown during the execution. In fact, divergences happen if an element existing at crawling time is no longer found at test time, when the application state is different, so that the desired path cannot be followed. The missing element triggers a *Selenium* exception.

RQ_4 (Coverage). Regarding coverage, which represents the core objective of test generation, we consider the transition coverage adequacy criterion, measured in the navigation graph specified by testers through POs. This required to manually map states and transitions in the crawled navigation graph to states and transitions in the PO navigation graph. We use statistical tests [AB14] to assess whether the difference between the two coverage medians is significant from the statistical point of view (Mann-Whitney U test [Kor04]).

For the sake of fairness, we granted both tools, SUBWEB and Ext-Crawljax, an overall execution budget of 2 hours and we ran both tools on the same subject 10 times, because both tools have non deterministic behaviour. In SUBWEB we have disabled the minimization step of Evosuite, because it requires multiple, costly test case executions on the browser, which makes it too inefficient for our purposes. Moreover, at every test case execution, we reset the state of the web application, both client and server side. In Ext-Crawljax, we use the default configuration with the default parameter values. We only provide Ext-Crawljax with custom values for those form inputs in the application that require very specific values. Ext-Crawljax does not reset the state of the application at every test case execution during crawling; however, we do reset the state of the web application when test are extracted from the crawled model.

4.5.3 Results

RQ_1 (Cost): The data in Table 4.1 shows that the 13 POs written manually account for 764 LOC in total. This is a small fraction of the overall application size (around 2%). Guards, that

Table 4.2: RQ_2: size of PO vs crawled graph, with missing/split states/transitions

	# States	# Trans	# Missing States	# Missing Trans	Split State Ratio	Split Trans Ratio
PO Graph	12	73	0	0	0	0
Crawled Graph	329	927	0	5	27	13

are required exclusively by our approach, represent an even smaller portion of the application size: guard method LOC account for 0.2% of the application size, while the 16 guards use on average 3 logical operator each. Moreover, we wrote the 13 POs in, approximately, one day. However, this metric clearly depends on many factors, the main one is the level of confidence the developer has with the PO pattern.

Overall, based on the size data collected on our case study, the manual cost for writing POs and PO guards seems relatively low.

RQ_2 (Navigation graph): As shown in Table 4.2, the crawled navigation graph is huge if compared to the PO navigation graph (approximately $\times$27 states; $\times$13 transitions). While it does not miss any state, despite its size it misses on average 5 transitions, which are specified by testers, but are not covered during some executions of crawling (5 is the average computed over 10 runs of Ext-Crawljax). No single case of state/transition merge was observed, while, as expected from the larger graph size, several states and transitions are split in the crawled graph.

The crawled graph deviates from the ideal, manually specified, PO graph to a major extent, because of its larger size, missing transitions and split states/transitions.

RQ_3 (Test Suite Features): Table 4.3 shows that SUBWEB generates much smaller test suites than Ext-Crawljax. This is a consequence of the different navigation graph size. Moreover, while SUBWEB generates non divergent test cases by construction, the crawling based approach generates as many as 17% divergent test cases.

The test suites produced by SUBWEB *are approximately* 11 *times smaller than the test suites produced by Ext-Crawljax. The latter include a relatively large proportion of divergent test cases.*

RQ_4 (Coverage): Figure 4.10 shows the box plots of the transition coverage achieved by SUBWEB and Ext-Crawljax. Table 4.3 also shows that the median coverage of SUBWEB is on average 13pp (percentage points) above the median coverage of Ext-Crawljax and such a difference is statistically significant according to the Mann-Whitney U test [Kor04] (p-value $4.7 \cdot 10^{-4}$

Table 4.3: RQ_3: number of test cases and divergent test cases in SUBWEB and Ext-Crawljax. RQ_4: transition coverage (%) achieved by SUBWEB and Ext-Crawljax in 10 runs; the two distributions differ in a statistically significant way according to the Mann-Whitney U test [Kor04] (p-value $4.7 \cdot 10^{-4}$).

	RQ_3: Test Suite Features			RQ_4: Transition Coverage (%)	
	# Tests	# Divergent Tests	% Divergent Tests	Median	Variance
SUBWEB	54	0	0	96	12
Ext-Crawljax	598	104	17	83	39

at 5% significance level), that we applied since we didn't have a priori knowledge about the distribution of the data.

The test cases generated by SUBWEB *achieve higher transition coverage than those generated by Ext-Crawljax.*

4.5.4 Threats to Validity

Threats to the *internal validity* might come from how the empirical study was carried out. Each test case was run starting from an empty database, under the assumption that the tester is interested in the behaviour of the application when no record has been persisted yet. On the contrary, if a non empty database is created at each test case startup, the traversal of paths for which populating the database is a prerequisite becomes easier for both approaches. Moreover, we didn't use a case study with existing POs and measured the effort needed to modify them in order to enable our technique; indeed it is difficult to find open source projects with existing selenium tests using the PO design pattern. Furthermore, the reimplementation of Atusa's extraction algorithm constitutes an internal validity threat. However, we followed the algorithm description reported in the Atusa's paper [MvD09] for our reimplementation.

Threats to the *external validity* mainly regard the use of only one case study, which prevents us from generalizing our findings to substantially different cases. On the other hand, AddressBook is a non trivial application that has been used in several previous works on web testing.

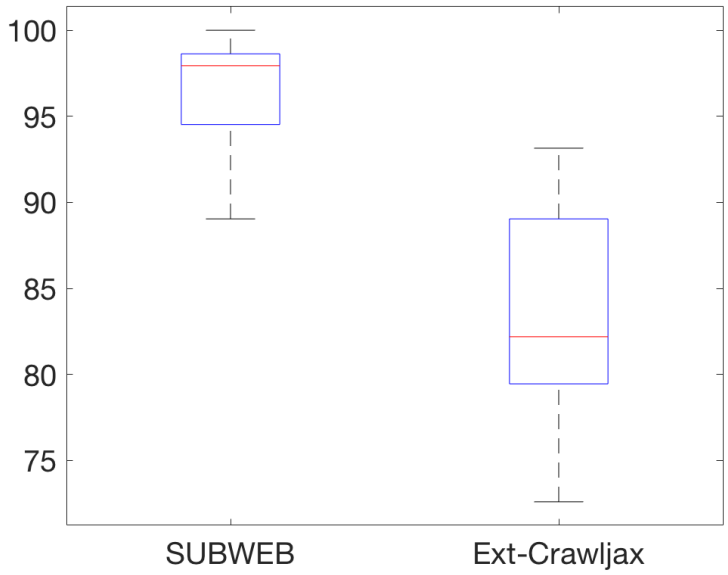


Figure 4.10: RQ_4: box plots of transition coverage achieved by SUBWEB and Ext-Crawljax

4.6 Limitations of Search Based Web Test Generation

Search based techniques iteratively sample the input space, selecting the fittest candidate test cases, and evolving the fittest ones using genetic search operators to create new test cases [Luk13]. Since these algorithms can effectively guide the generation of test cases even for large input spaces, they are suited for system-level testing [Zel17]. In the previous section, we showed that an effective fitness function can be defined for E2E web tests based on approximate information available in the navigation graph, specifically the actions guards, by making them explicit in the PO methods. Results show that this approach can guide the search towards generating test cases unaffected by the path infeasibility problem.

However, this approach needs the manual specification of all guards for each action, a task that is time-consuming and laborious for testers. Indeed, such information depends on the web application business logic and intended behaviour, and thus cannot be generated fully automatically. Additionally, the evaluation of the fitness function is costly, because it requires a large number of test cases to be generated and executed in the browser before converging to an adequate set of tests.

Moreover, while search based approaches provide theoretical warranties of asymptotic convergence to the desired input values, they exhibit poor execution time performance when applied to web applications, as compared, for instance, to standard Java desktop applications [FA13]. The reason why search based approaches are computationally expensive is that they always need to

execute the candidate test cases to assess their feasibility and the value of the fitness function. This disadvantage is amplified in web testing because (1) the overhead imposed by browser's interaction makes evaluating these tests very slow, and (2) the input space may contain local optimum regions, in which all candidate test cases are likely to be behaviourally equivalent (*e.g.*, all equally infeasible), hence increasing the run time of the test generation algorithm without benefiting the overall coverage. The discussed limitations (i.e., high computational costs, need for manual guards specification) justify the investigation of alternative, possibly cheaper and more automated algorithms.

4.7 Diversity Based Web Test Generation

Inspired by adaptive random testing [CLM04], which makes the assumption of *contiguous failure regions*, we conjecture that web applications might also have **contiguous infeasibility regions**. Correspondingly, in the case of web apps the *main advantage* of test case diversity would be the possibility of exploring the search space at large, diversifying the region where navigation sequences are sampled. This is expected to help escaping local solutions, as well as avoiding generating infeasible test paths and ensuring a more effective exploration of alternative (*i.e.*, *diverse*) behaviours. Indeed, research has shown that diversity is especially beneficial to fault detection. Diverse inputs are necessary to expose different failures, whereas inputs from contiguous areas of the input space are likely to expose the same program failure [CKMT10].

A *second advantage* of diversity based test case generation is its higher efficiency with respect to existing random and search based approaches. In fact, the quality of a candidate test case is evaluated by measuring its *diversity* with respect to previously generated test cases and using such metric to assess its potential at increasing the exploration of diverse behaviours. Interestingly, such assessment can be performed without actually executing the candidate test cases.

Algorithm 2 describes our overall procedure for diversity based path and input generation. The test generation starts from an empty set of tests, and therefore the first generated test is random. The algorithm generates a set C of *candidate* test cases, instantiating candidate test paths along with concrete input vectors (Lines 8–10). To select the most promising candidates, the distance between each candidate test case and the current set of already executed test cases T_{exec} is computed (Lines 11–17) and only the farthest test case t is executed (Line 18). Test case t is restricted to its feasible prefix in case it includes a divergence (Lines 19–21). A *divergence* occurs whenever the test path of a test case t differs from the execution trace obtained by running t. Then, the test case is added to the final test suite TS_{gen} only if it contributes to increase coverage of the navigation graph $\mathcal{G}$ (Lines 22–24).

Algorithm 3 shows how candidate test paths are created. In our notation, S represents a state sequence, A indicates a method sequence having X as a input vector sequence. S, A and X are incrementally created within the main loop (Lines 5–11) by choosing an edge $\langle n, n' \rangle_{[r]e}$ ran-

Algorithm 2: Diversity-based Test Case Generation

Input : $\mathcal{G}$: navigation graph, k: number of candidates
Output: TS_{gen}: test suite that optimizes coverage of $\mathcal{G}$

1 $T_{exec} \leftarrow \emptyset$ ▷ *Set of executed test cases*
2 $TS_{gen} \leftarrow \emptyset$ ▷ *Set of generated test suite*
3 generate randomly a test case t, add t to T_{exec}, and execute it
4 **while** $\mathcal{G}$ *is not adequately covered by* TS_{gen} *or timeout is not reached* **do**
5 $\quad e \leftarrow$ GETRANDOMUNCOVEREDMETHOD($\mathcal{G}$)
6 $\quad D_{max} \leftarrow 0$
7 $\quad C \leftarrow \emptyset$ ▷ *Set of candidate test cases*
8 $\quad$ **for** $i \leftarrow$ *1 to* k **do**
9 $\quad\quad C \leftarrow C \cup$ GENERATECANDIDATETEST($\mathcal{G}$, e) ▷ *see Algo 2*
10 $\quad$ **end**
11 $\quad$ **for** $c_i \in C$ **do**
12 $\quad\quad d_i \leftarrow min(\rho(c_i, t_j)) \quad \forall t_j \in T_{exec}$ ▷ *see Eq 1*
13 $\quad\quad$ **if** $d_i > D_{max}$ **then**
14 $\quad\quad\quad D_{max} \leftarrow d_i$
15 $\quad\quad\quad t \leftarrow c_i$
16 $\quad\quad$ **end**
17 $\quad$ **end**
18 $\quad$ add t to T_{exec} and execute it
19 $\quad$ **if** t *is divergent* **then**
20 $\quad\quad t \leftarrow$ GETFEASIBLEPREFIX(t)
21 $\quad$ **end**
22 $\quad$ **if** t *increases coverage of* $\mathcal{G}$ *w.r.t.* TS_{gen} **then**
23 $\quad\quad TS_{gen} \leftarrow TS_{gen} \cup \{t\}$
24 $\quad$ **end**
25 **end**
26 **return** TS_{gen}

domly, with uniform probability, among those available from state n according to the navigation graph $\mathcal{G}$. The selection is constrained by the fact that the target method e must remain reachable from n'. At last, a vector of input values as parameters to e is randomly chosen (Line 9).

4.7.1 Distance Between Test Cases

Algorithm 2 requires a distance metric to assess the diversity between test cases. Differently from object-oriented [LTCZ09, CLOM08] or numerical applications testing [CKMT10], in our setting we cannot rely only on the input values, as the distance function ρ must also take into account the sequence of actions composing a test case. As such, we devised a novel distance metric between two test cases taking into account the *diversity of the respective sequences of actions*, and, in cases in which this is not discriminative, privileging the test case having the farthermost diversity in terms of concrete input values. By diversifying the sequence of actions, as well as

Algorithm 3: Candidate Test Case Generation

Input : $\mathcal{G}$: navigational model, e: target method
Output: $\langle S, A, X \rangle$: candidate test case reaching e

$A \leftarrow \langle\rangle$
$X \leftarrow \langle\rangle$
$n \leftarrow$ GETROOTNODE($\mathcal{G}$)
$S \leftarrow \langle n \rangle$
while $e \notin A$ **do**
  $\langle n, n' \rangle_{[r]e} \leftarrow$ GETRANDOMEDGE($\mathcal{G}$, n, e) ▷ *must ensure* $n' \rightsquigarrow e$
  $n \leftarrow n'$
  $x \leftarrow$ GETRANDOMINPUTVECTOR(e)
  $S \leftarrow S$.*add*(n)
  $A \leftarrow A$.*add*(e)
  $X \leftarrow X$.*add*(r)
end
return $\langle S, A, X \rangle$

the associated input values, we conjecture that we can escape infeasible test path regions and we can diversify the WAUT behaviours.

4.7.1.1 Distance Formula

Intuitively, our distance formula comprises two terms, the first measuring the distance between the action sequences in the two test cases being compared and the second measuring the distance between the input values used by matching actions in the two sequences. To compute the first term, the distance function α reports the number of non-matching actions in the two sequences. To compute the second term, we rely on the longest common subsequence to obtain the matching actions; upon each matched actions, we compute the normalized distance β between their parameter values.

Given two test cases (t_i, t_j), where $t_i = \langle S_i, A_i, X_i \rangle$ and $t_j = \langle S_j, A_j, X_j \rangle$, the distance $\rho(t_i, t_j)$ is given by Equations (4.1), (4.2):

$$\rho(t_i, t_j) = \alpha(N_i :: A_i, N_j :: A_j) \tag{4.1}$$

$$+ \sum_{\langle k_1, k_2 \rangle \in LCS_{i,j}} \beta(X_i[k_1], X_j[k_2])$$

$$\beta(x, y) \;=\; \frac{1}{\mid x \mid} \sum_{q \in [1 \ldots |x|]} \delta(x[q], y[q]) \tag{4.2}$$

In Equation 4.1, the notation $N_x{::}A_y$ indicates that the two sequences of actions are identified both by their actions A_i, A_j and the states N_i, N_j in which they are applicable. This helps disambiguate cases in which an action A_y is present with the same name in different states.

Function α represents the *sequence edit distance* [CLRS01, Lev66], which determines the number of non-matching elements in two sequences (*e.g.*, 3 = $\mid \langle a \rangle \mid + \mid \langle e, f \rangle \mid$, when comparing $\langle a, b, b, c \rangle$ to $\langle e, b, c, f \rangle$). LCS is the set of matching indexes in the *longest common subsequence* [CLRS01, Lev66] (*e.g.*, $\{(2, 2), (4, 3)\}$, when comparing $\langle a, b, b, c \rangle$ to $\langle e, b, c, f \rangle$). Function β computes the distance between the two parameter value sequences $X_i[k_1], X_j[k_2]$ of two matching actions (indexed by k_1 and k_2 respectively in $N_i{::}A_i$ and $N_j{::}A_j$). At last, function δ computes the normalized distance between two primitive input values of the same type (see Table 4.4).

The second term of Equation 4.1 matches the actions $A_i[k_1], A_j[k_2]$ in the two sequences. Correspondingly, the two input vectors $X_i[k_1]$ and $X_j[k_2]$ have the same cardinality. Function β computes the average distance between the parameter values x and y of two matching actions. It resorts to function δ to compute the normalized distance between pairs of primitive input values $x[k]$, $y[k]$. The computation of function δ varies depending on the type of parameters occurring into the actions (see Table 4.4). The input distance δ is normalized between 0 and 1. For types `string` or `number`, normalization is achieved using function $\eta(x) = \frac{x}{x+1}$ as proposed by Arcuri *et al.* [Arc13], that maps a value x onto the range $[0, 1]$.

Equation 4.1 resembles the computation of the *fitness function* for branch coverage, commonly adopted in search-based testing [McM04]. In such a case, the two terms of the fitness function are *approach level* and normalized *branch distance*, respectively. We share with that definition the idea of making the first term (action sequence distance) dominant, while resorting to the second term (input value distance) only when the first term is not discriminative enough. In fact, the range of α is $\mathbb{N}$, so the minimum non-zero distance is 1, whereas β ranges between 0 and 1, such

Table 4.4: Input distance computation

Type	**Input Distance** $\delta(w, z)$
`boolean`	$\delta(w, z) = \{0$ if $w = z$; 1 otherwise $\}$
`enum`	$\delta(w, z) = \{0$ if $w = z$; 1 otherwise $\}$
`string`	$\delta(w, z) = \alpha(w, z)/(1 + \alpha(w, z))$
`number`	$\delta(w, z) = \lvert w - z \rvert / (1 + \lvert w - z \rvert)$

that the contribution to ρ of each matching pair of actions cannot be greater than 1. The intuition is that the major contribution to diversity comes from the action sequence distance, while the input value distance for each matching action pair contributes at most as the minimum non-zero action sequence distance (*i.e.*, 1). However, in case of a large number of actions being matched, the aggregate value of the input value distance might become dominant.

4.7.2 Example of Distance Computation

We present how the distance between test cases is computed using a simplified notation in which only the actions are reported while omitting the states (*e.g.*, `Index::login` becomes `login`). In fact, in all our examples the state sequence associated with a given action sequence will be unique (this is not true in the general case). Moreover, parameter values are indicated in brackets rather than separate input vectors (*e.g.*, an action sequence $A = \langle$`login`$\rangle$ and the corresponding input sequence $X = \langle\langle$`"credentials"`$\rangle\rangle$ become $\langle$`login("credentials")`$\rangle$.

Let us consider three simple test cases t_1, t_2, t_3 for the running example Phoenix Trello, defined as follows:

t_1 = $\langle$`login("credentials1"), signOut()`$\rangle$
t_2 = $\langle$`login("credentials2"),addNewBoard("board"),goToBoardsPage()`$\rangle$
t_3 = $\langle$`login("credentials2"), signOut()`$\rangle$

Suppose that t_1 is already in the list of generated test cases T_{gen}, whereas t_2 or t_3 are in the set of generated tests C. Our algorithm must decide which test to execute next. The sequence edit distance between t_1 and t_2 is α = 3, because there are three non-matching actions in t_1, t_2 (`signOut` in t_1, and `addNewBoard`, `goToBoardsPage` in t_2).

Concerning the matching actions (`login` in both t_1 and t_2), the input distance is computed by function β, which in turn resorts to function δ for the distance between primitive values. In our running example, β((`"credentials1"`), (`"credentials2"`)) = δ(`"credentials1"`, `"credentials2"`) = 1/2 = 0.5, because α(`"credentials1"`, `"credentials2"`) = 1 (the one non matching character in `"credentials2"` being '2'). Therefore, $\rho(t_1, t_2)$ = 3 + 0.5 = 3.5.

Conversely, the sequence edit distance between t_1 and t_3 is α = 0, since there are no unmatched actions in the two sequences. Therefore, our distance formula relies on the input distance β((`"credentials1"`), (`"credentials2"`)) = 1/2. Therefore, $\rho(t_1, t_3)$ = 0 + 1/2 = 0.5. Thus, the *most diverse* test case t_2 is selected and executed.

4.7.3 Implementation

We implemented our approach in a Java tool called DIG (**DI**versity-based **G**enerator), which is publicly available [4]. To retrieve the PO testing model, DIG relies on Apogen [SLRT17]. Finally, our diversity-based test generator is implemented on top of Evosuite [FA13], which we extended to handle the PO model. Our algorithm uses the PO method information to generate and evaluate candidate test cases. At last, only the most diverse candidates are executed through Selenium WebDriver in *headless* execution mode.

4.8 Empirical Evaluation

The goal of the empirical evaluation is to study the effectiveness and the efficiency of the proposed diversity based approach. In particular, we want to investigate empirically our conjecture that by reducing the number of executions with respect to the existing web test generators, our diversity based approach allows a better exploration of the navigation graph. As baselines, we consider the test cases that can be extracted from the navigation graph produced by a state of the art crawler, Crawljax [MvDL12, MvD09] (i.e. our reimplementation of Atusa [MvDR12] which we call Ext-Crawljax) and the test cases that are generated by our search based web test generator SUBWEB, presented in Section 4.4.

4.8.1 Research Questions

We address the following research questions:

RQ_1 (Effectiveness): *How do diversity, search, and crawling based random test generation compare in terms of transition coverage, code coverage and fault detection?*

RQ_2 (Efficiency): *How do diversity and search based test generation compare in terms of efficiency over time?*

RQ_3 (Distance Computation): *What is the impact of distance computation in the diversity based test generation process?*

RQ_4 (Manual POs): *What is the effect of using manually defined POs within the diversity and search based test generation approaches?*

RQ_1 and RQ_2 aim to compare DIG with two state-of-the-art solutions: our previous search based web test generator called SUBWEB and Ext-Crawljax, which is based on a crawling-based random approach. RQ_3 and RQ_4 aim at assessing the impact of the internal factors of the

[4] https://github.com/matteobiagiola/FSE19-submission-material-DIG

proposed approach (namely, distance computation and POs generation method) on the final test suites.

4.8.2 Subject Systems

To assess the relevance of our approach at testing real-world web applications, we focus our analysis on single page applications (SPA). We overviewed the most popular JavaScript frameworks for developing web applications from *GitHub Collections* [JS-18]. Popularity was measured as the number of stars owned by the framework's GitHub repository at the time of writing (August 2018). We retained five frameworks with more than $15k$ stars.

Second, we selected web applications developed with one of the selected frameworks that are popular (number of stars ≥ 50), mature (number of commits ≥ 50) and have been maintained recently (year of last commit ≥ 2016). Third, from the resulting candidate set, in order to maximize diversity and representativeness, we randomly sampled six applications considering the six most popular JavaScript frameworks: dimeshift (Backbone.js), pagekit (Vue.js), splittypie (Ember.js), phoenix-trello (Phoenix/React), retroboard (React), PetClinic (AngularJS).

Table 4.5 summarizes the main characteristics of our subjects. The size of the selected systems ($> 1k$ client-side JavaScript LOCs, frameworks excluded) is representative of modern web applications [OJPM17] (Ocariza *et al.* [OJPM17] report an average of 1,689 LOCs for a dataset of web applications developed with the AngularJS framework with at least 50 stars).

We focused on SPA because, in such category of web applications, the business logic is moved towards the client side. Only in such context it is reasonable to compare the competing test generators by measuring their client side code coverage, since the ultimate objective of functional testing is to exercise the functionalities (hence the logic) of the web application under test. Moreover, client side code coverage is easier to measure than server side code coverage in E2E GUI testing, both because JavaScript is the only language used for client side development (w.r.t

Table 4.5: Experimental subjects

Subject	Framework	LOC (JS)	Stars	Commits	Year
dimeshift [dim18]	Backbone	5,140	127	194	2018
pagekit [pag18]	Vue.js	4,214	4,851	4,914	2019
splittypie [spl18]	Ember.js	2,710	67	331	2017
Phoenix Trello [pho18]	React	2,289	2,233	422	2016
retroboard [ret18]	React	2,144	390	476	2019
PetClinic [Pet18]	AngularJS	2,939	50	71	2018

many server side programming languages) and because existing unit testing JavaScript coverage tools can be *easily* adapted to E2E testing JavaScript coverage tools.

4.8.3 Procedure and Metrics

To answer the research questions RQ_1, RQ_2, RQ_3 we generated the POs needed for DIG and SUBWEB with the tool Apogen [SLRT17]. To answer RQ_4 we wrote the POs manually for each subject. From the POs we extract the navigation graph with Algorithm 1 and we reformulate the problem of transition coverage of the navigation graph as a branch coverage problem by using the program transformation procedure described in Figure 4.6.

Effectiveness (RQ_1). We ran DIG, SUBWEB and Ext-Crawljax on each subject system. For DIG, we set the number of candidate test cases generated at each step of the algorithm to 50. As SUBWEB requires, we manually specified the guards for the POs methods. For DIG and SUBWEB, at every test case execution, we reset the state of the web application under test, both client and server side. This way the generated tests are independent. Ext-Crawljax does not reset the state of the application at every test case execution during crawling; however, we do reset the state of the web application when test are extracted from the crawled model. We granted each tool the same time budget of 30 minutes because, in our exploratory experiments, we empirically observed convergence of transition coverage to a plateau within half an hour. Additionally, we repeated each experiment 15 times, and computed the average across all executions to cope with non-deterministic behaviours.

We considered three metrics of effectiveness. First, we measured the *transition coverage* of the navigation graph according to the transition coverage adequacy criterion. This metric tells us how many functionalities each approach covers. Second, we measured *branch coverage* of the JavaScript code of our subject systems. We instrumented the client side of each web application (libraries and framework excluded) with the tool `Istanbul` [Ist18], and executed the generated test suites against the instrumented applications. Code coverage is an important metric to understand how much of the actual business logic is exercised by each approach. Third, we studied the *fault detection* capability, by counting the number of unique faults (*i.e.*, unique JavaScript exceptions and errors) reported in the JavaScript console upon test suite execution. The number of unique faults let us understand the capabilities of a certain approach to expose potential errors of the web application under test.

Efficiency (RQ_2). To assess *efficiency*, we measured how the competing algorithms perform over time in terms of transition coverage, branch coverage, and fault detection. To this aim, we computed the area under the curve (AUC) of each metric as a function of the test generation time, with higher values of AUC denoting a superior efficiency of the algorithm within the given time budget (30 minutes).

Concerning transition coverage, we could compute the AUC accurately, because Evosuite outputs the value of such metric at every new test case generation. Differently, to measure AUC for branch coverage and fault detection, we had to execute each intermediate test suite produced during the test generation process. Given the huge number of test suites to be considered (in the order of dozens of thousands considering all applications, approaches and repetitions), for each subject, we sampled three time intervals: the point at which the transition coverage difference between DIG and SUBWEB is maximal, the point at which it is 50% of the maximum, and 30 minutes. In fact, by graphically plotting the two transition coverage functions, we observed transition coverage difference has a steep peak followed by a smooth decline. Hence, in order to accurately estimate the AUC with only a few data points (time intervals), it makes sense to sample them where the difference is the largest, and the next one, when it is halved.

Distance Computation (RQ_3). Distance computation is expensive because it grows quadratically with the number of test cases: at each test generation step, the number of distance computations is given by the number of previously executed test cases multiplied by the number of candidates. The input distance term of Equation 4.1 further increases the cost for distance computation. To assess such impact, we ran DIG disabling the input distance computation, thus computing the distance as just the sequence edit distance. We refer to these two variants of our approach as DIG_S and DIG_{S+I}, respectively, and computed the same effectiveness/efficiency metrics used for RQ_1 and RQ_2. Additionally, we assessed the overhead of distance computation on the number of test executions by reporting the number of tests generated and executed by each tool/configuration.

Manual POs (RQ_4). A developer may develop more accurate POs than those produced by an automatic technique. Thus, we compared the effectiveness and efficiency of DIG and SUBWEB when manually defined POs are utilized. We wrote the POs for each subjects following the rigorous procedure presented in Section 4.5.2 (see RQ_1 (Cost)).

To compare automatically *vs* manually generated POs, we computed the same effectiveness and efficiency metrics used to answer RQ_1 and RQ_2, as described above.

Table 4.6: Effectiveness, Efficiency and Distance Computation results (Automated POs) for RQ_1, RQ_2, and RQ_3 for all subjects and approaches. Values in bold indicate statistically significant differences between DIG and SUBWEB. Stars indicate statistically significant differences between DIG_{S+I} and DIG_S. NC indicates non-comparable values since the test generation terminates before the given time budget.

	EFFECTIVENESS																		EFFICIENCY										DISTANCE COMPUTATION							
	Structural Coverage							**Faults**				**Structural Coverage**						**Faults**				**Tests**					**Distance**							
	Trans Cov. (%)			*Branch Cov. (%)*				*Avg Unique (#)*				*Trans AUC (%)*			*Branch AUC (%)*			*AUC (%)*			*Gen.*			*Exec.*		*#*							
	DIG_{S+I}	DIG_S	SUBWEB	DIG_{S+I}	DIG_S	SUBWEB	Ext-Crawljax	DIG_{S+I}	DIG_S	SUBWEB	Ext-Crawljax	DIG_{S+I}	DIG_S	SUBWEB	DIG_{S+I}	DIG_S	SUBWEB	DIG_{S+I}	DIG_S	SUBWEB	DIG_{S+I}	DIG_S	SUBWEB	DIG_{S+I}	DIG_S	DIG_{S+I}	DIG_S							
dimeshift	99.4	**100.0**	98.4	40.5	40.4	40.3	18.8	2.7	**3.3**	2.5	1.0	**97.2**	**97.2**	95.1	**36.2**	35.9	34.9	**50.0**	**56.2**	35.2	NC	NC	NC	NC	NC	NC	NC							
pagekit	96.3	97.6	96.3	27.9	27.2	28.0	24.8	1.0	1.1	1.1	0.0	**94.5**	**96.2**	93.3	24.6	24.6	26.6	33.0	34.6	41.7	69.5k	7.9k	218	158	218	487k	628k							
splittypie	94.0	93.8	91.0	51.9	51.7	50.1	18.6	5.6	6.0	5.6	1.0	**91.0**	**90.7**	86.3	**47.3**	**45.8**	44.3	84.5	89.6	86.2	11k	11.8k	329	221	236	1,227k	1,398k							
phoenix	99.7	99.7	99.2	64.1	62.8	62.5	34.2	3.1	**3.5**	3.1	0.0	**98.1**	**98.1**	94.8	**58.0**	**58.7**	55.4	**58.1**	**66.2**	47.8	NC	NC	NC	NC	NC	NC	NC							
retroboard	100.0	100.0	100.0	71.3	71.7	69.8	51.4	0.0	0.0	0.0	0.0	**97.9***	96.1	96.3	**69.1**	**68.7**	67.4	0.0	0.0	0.0	NC	NC	NC	NC	NC	NC	NC							
PetClinic	100.0	100.0	100.0	85.0	85.0	85.0	0.0	2.5	2.9	2.1	0.0	96.4	95.8	97.3	**79.1**	75.7	68.7	41.0	**45.6**	32.5	NC	NC	NC	NC	NC	NC	NC							
Average	99.7	98.5	97.5	56.8	56.5	56.0	24.6	2.5	2.8	2.4	0.3	95.9	95.7	93.9	52.4	51.6	49.6	44.4	48.7	40.6	40.2k	15.3k	274	180	197	857k	1,000k							

4.8.4 Results

Effectiveness (RQ_1). Table 4.6 (*Effectiveness*) compares DIG_{S+I}, DIG_S, and SUBWEB in terms of transition coverage, code coverage and fault detection. Statistical significance of the difference was assessed by applying the non-parametric Mann-Whitney U test [Kor04], with a confidence threshold $\alpha = 0.05$.

Concerning Ext-Crawljax, we do not report comparisons in terms of transition coverage, because the navigation graph retrieved by the crawler is different from those based on the PO abstraction that DIG generates. To compare them accurately, one would need to find the isomorphism between the two navigation graphs, which requires mapping each state and each transition from one model onto the other—a manual and expensive process. Actually, we performed such mapping when we evaluated the search based approach in the previous section (see Section 4.5.2) and found a substantially higher coverage of SUBWEB w.r.t Ext-Crawljax. For those reasons, for Ext-Crawljax, we only compare code coverage and fault detection metrics.

Looking at the results, both DIG and SUBWEB outperform Ext-Crawljax substantially. In all cases the differences are statistically significant (not reported in Table 4.6). As far as effectiveness is concerned, when automated POs are utilized, DIG and SUBWEB can be considered comparable test generators, with minimal performance variations. Despite both tools cover almost all navigation graphs within the given time budget of 30 minutes, there is however a remarkable difference. DIG is totally automated, and achieved these results by relying only on its diversity heuristic. SUBWEB, on the contrary, is semi-automatic, as it takes advantage of manually-defined guards to guide the search and avoid path infeasibility.

Diversity and search based approaches achieve substantially higher transition coverage, code coverage and fault detection than the crawling based random approach. Despite being comparably effective, the diversity based approach is preferable because it is fully automated.

Efficiency (RQ_2). Table 4.6 (*Efficiency*) compares DIG and SUBWEB in terms of transition coverage, code coverage and fault detection achieved over time (AUC metrics). DIG outperforms SUBWEB by a statistically significant amount in 5/6 subjects for transition and branch coverage. Regarding fault detection, DIG is significantly better than SUBWEB in 3/5 subjects (*retroboard* revealed no faults).

The plot in Figure 4.11 shows a meaningful example regarding the efficiency difference on transition coverage for *Phoenix Trello*. DIG reaches the maximum transition coverage after nearly one third of the total time budget, whereas SUBWEB takes approximately twice as much time. Moreover, for small time intervals, the effectiveness difference of DIG *vs* SUBWEB is further amplified. For instance, the maximum difference between the two algorithms is 22% after 2 minutes ($\approx$6% of the time budget). In practice, this means that DIG is preferable if strict testing time constraints apply.

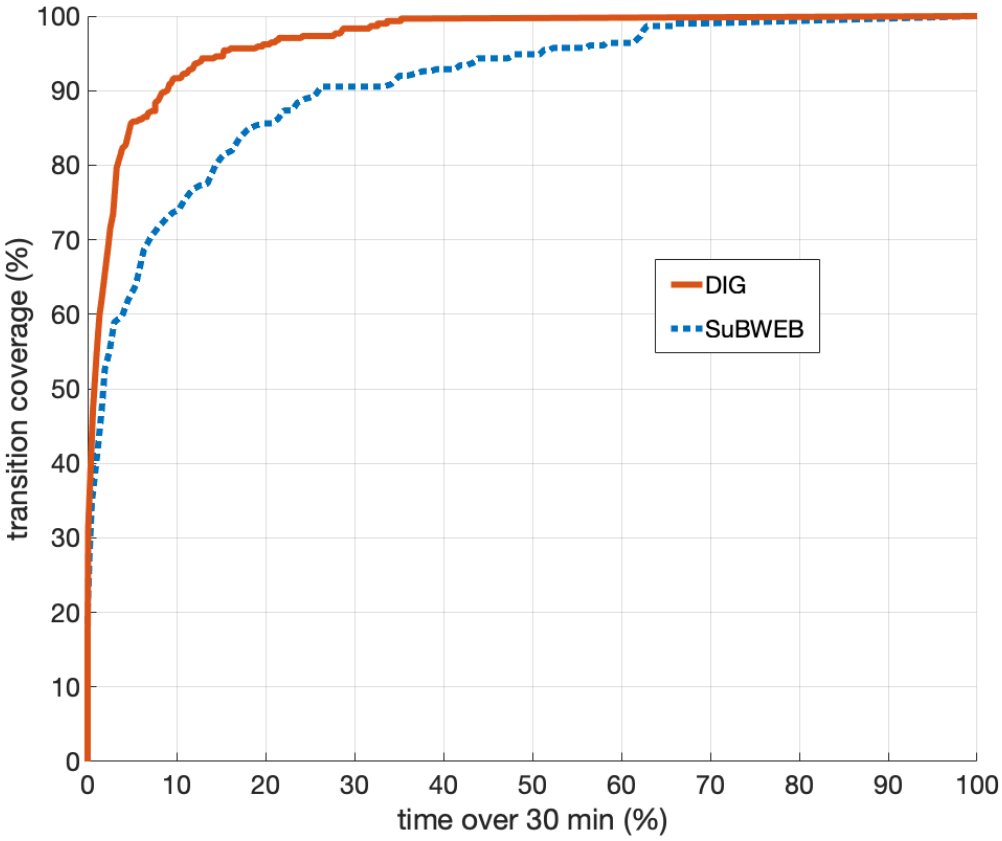


Figure 4.11: Average efficiency over time in terms of transition coverage of the compared approaches on *Phoenix Trello*

.

The diversity based approach achieves high coverage and fault detection rates substantially faster than the search based approach.

Distance Computation (RQ_3). Table 4.6 (*Distance Computation*) shows the number of test cases generated and executed by DIG and SUBWEB and the number of distance computations required by DIG_{S+I} and DIG_S (for SUBWEB the number of generated test cases always equals the executed test cases).

On average, DIG computed a large number of distances ($857k$ by DIG_{S+I} and $1,000k$ by DIG_S). Such computations reduce the time available for test case execution. Thus, while SUBWEB runs on average 274 test cases, DIG_{S+I} and DIG_S run an average of 94 and 77 less test cases, respectively (*dimeshift*, *Phoenix Trello*, *retroboard* and *PetClinic* are excluded from the analysis

Table 4.7: Manually vs Automatically generated POs

	PO Size (LOC)			# PO Methods		
	Apogen	Manual	Incr (%)	Apogen	Manual	Incr (%)
dimeshift	511	760	49	35	72	105
pagekit	567	2,030	258	42	214	409
splittypie	492	560	14	31	44	42
Phoenix Trello	324	482	49	24	38	58
retroboard	292	350	20	26	29	11
PetClinic	312	450	44	20	47	135
Average	416	772	72	30	74	127

since they always terminate before 30 minutes, which means that the number of test executions is not constrained by the time budget).

The number of generated test cases (not necessarily executed) by DIG is substantially higher than the test cases generated and executed by SUBWEB (e.g., $40.2k$ and $15.3k$ *vs* 274). However, despite a lower number of test executions, the time spent in distance computation allows DIG to produce test cases that have a higher chance of increasing coverage and fault detection (see results for **RQ$_1$** and **RQ$_2$**). This confirms our initial hypothesis that it is possible to assess the quality of web test cases without executing them, while still achieving high coverage and fault detection rates.

Let us now compare DIG$_{S+I}$ and DIG$_S$. The extra computational cost associated with the input distance reduces on average the number of executed test cases by $\approx$9%. On the other hand, the increased accuracy of the distance computed by DIG$_{S+I}$ does not bring considerable advantages in terms of coverage or fault detection (see results for **RQ$_1$** and **RQ$_2$**).

The overhead brought by the distance computation is more than compensated by the benefits in efficiency and automation. Moreover, the input component of the proposed distance metric can be discarded with little to no associated penalty.

Manual POs (RQ$_4$). Table 4.8 presents the results obtained when manually defined POs are utilized for test generation. For all subjects, both tools did not cover the entire navigational model within the given time budget (30 minutes). This can be explained by the higher number of methods contained in the manual POs, which model the web applications more accurately, at the cost of making test generation more challenging.

Table 4.8: Effectiveness, Efficiency and Distance Computation results (Manual POs) for RQ_4, and RQ_3 for all subjects and approaches. Values in bold indicate statistically significant differences between DIG and SUBWEB. Stars indicate statistically significant differences between DIG_{S+I} and DIG_S.

	EFFECTIVENESS												EFFICIENCY			DISTANCE COMPUTATION						
	Structural Coverage							**Faults**				**Structural Coverage**			**Tests**					**Distance**	
	Trans Cov. (%)			*Branch Cov. (%)*				*Avg Unique (#)*				*Trans AUC (%)*			*Gen.*		*Exec.*			*#*	
	DIG_{S+I}	DIG_S	SUBWEB	DIG_{S+I}	DIG_S	SUBWEB	Ext-Crawljax	DIG_{S+I}	DIG_S	SUBWEB	Ext-Crawljax	DIG_{S+I}	DIG_S	SUBWEB	DIG_{S+I}	DIG_S	SUBWEB	DIG_{S+I}	DIG_S	DIG_{S+I}	DIG_S
dimeshift	**70.8**	**70.3**	67.2	**41.7**	**41.3**	40.1	18.8	2.7	3.1	2.8	1.0	**67.4**	**66.9**	62.2	31.4k	27.0k	628	541	490	6,015k	7,331k
pagekit	**65.0**	**65.1**	61.6	34.5	**35.3**	33.3	24.9	2.7	2.8	2.7	0.0	**60.4**	**60.6**	56.4	9.7k	7.3k	196	146	142	965k	537k
splittypie	**74.7**	**74.4**	67.0	**51.7**	**51.4**	50.6	18.7	7.0	6.9	6.7	1.0	**70.8**	**70.6**	61.4	13.7k	12.5k	275	250	225	1,271k	1,569k
phoenix	**72.1**	**74.6**	68.8	63.4	**64.4**	61.8	34.2	**2.8**	2.3	1.4	0.0	**69.5**	**72.4***	65.5	16.5k	13.6k	331	272	243	1,482k	1,856k
retroboard	85.9	**88.1**	85.3	82.2	83.1*	82.3	51.5	0.0	0.0	0.0	0.0	**84.0**	**86.0***	81.5	23.0k	19.8k	462	396	352	3,106k	3,930k
PetClinic	**69.2**	**69.8**	65.7	49.1	46.8	44.1	0.0	**3.7**	3.1	3.0	0.0	**66.9**	**67.2**	62.2	12.4k	12.6k	248	253	356	1,544k	1,607k
Average	73.0	73.7	69.3	53.8	53.7	52.0	24.7	3.2	3.0	2.8	0.3	69.8	70.6	64.9	18.7k	15.5k	375	310	283	2,397k	2,805k

Table 4.7 shows information about the size, in lines of code (LOC), of the POs of our study, as well as the number of methods they contain. Methods determine the transitions, hence the complexity, of the navigation graph. The manually generated POs contain, on average, 72% more LOC (Column 4) and 127% more transitions (Column 7) than Apogen's POs.

For 3/6 subjects (namely *dimeshift*, *Phoenix Trello*, and *PetClinic*), we observed no significant difference between using automated or manual POs in terms of branch coverage and fault detection (see results of DIG in Table 4.6 and Table 4.8). Overall, when manually defined POs are available, DIG outperforms SUBWEB with statistical significance in terms of transition coverage in all cases, and in 4/6 cases concerning branch coverage.

When using manually-defined POs, the diversity based approach outperforms the search based approach in terms of structural coverage reached by the generated test suites.

4.8.5 Threats To Validity

Using a limited number of subject systems in our evaluation poses an *external validity* threat, in terms of generalizability of our results. We tried to mitigate this threat by choosing six subject systems developed with real-world JavaScript frameworks and pertaining to different domains, although more subject systems are needed to fully address the generalization threat.

Threats to *internal validity* might come from confounding factors of our experiments. We compared all competing algorithms under identical parameter settings (e.g., time budget intervals), on real-world web applications. The manual PO development task poses a threat to validity that we tried to mitigate by following a rigorous, systematic procedure. Moreover, the reimplementation of Atusa's extraction algorithm constitutes an internal validity threat. However, we followed the algorithm description reported in the Atusa's paper [MvD09] for our reimplementation.

Conclusion validity is related to random variations and inappropriate use of statistical tests. To mitigate these threats, we ran each experiment 15 times and used the non-parametric Mann-Whitney U test for statistical testing which does not make any assumption about the distribution of the data.

With respect to *reproducibility* of our results, the source code of DIG and all subject systems are available online [5], making the evaluation repeatable and our results reproducible.

[5] `https://github.com/matteobiagiola/FSE19-submission-material-DIG`

4.8.6 Discussion

Effectiveness and Automation. By combining POs and test generation, diversity and search based approaches achieved substantially higher transition coverage, code coverage and fault detection than a state-of-the-art crawling based, PO agnostic approach.

DIG is fully automated while SUBWEB is only semi-automated as manual preconditions need to be specified. Our results show that DIG can potentially save testers a considerable amount of time by generating both POs and test cases automatically. If necessary, testers can refine the generated POs with missing actions or transitions, and repeat the test generation.

Additionally, the test suites generated by our approach are based on the page object design pattern, which brings known advantages in terms of maintainability [LCRT16].

Efficiency. Our efficiency results demonstrate that our diversity based approach is preferable, especially in settings where the time devoted to testing is strict or when test cases are executed very often during the development process. Indeed, the diversity based approach achieved high coverage and fault detection scores substantially earlier than the search based approach, regardless of the POs being used. This gives us confidence in the applicability of our technique in modern software development processes such as XP and DevOps.

Benefits to Feasibility. According to our empirical results, which confirmed the conjecture of contiguous infeasibility regions, promoting diversity is beneficial not only to a thorough exploration of the application behaviours, but also to the feasibility of automatically generated test cases. The search based approach, on the contrary, uses the guards in the navigational model explicitly to guide the search towards inputs that satisfy them. Therefore, DIG achieves feasibility as a side effect of diversity, while SUBWEB requires manual specification of guards to achieve feasibility.

Comparison with Manual POs. The POs automatically generated by Apogen are usually simpler than those developed manually, in terms of number and complexity of actions being exposed for testing. Apogen creates methods based on the actions statically extracted from each test state. As such, in most cases, the resulting methods may miss complex interactions that are possible on the web GUI. Additionally, due to a transition minimization strategy, Apogen does not create reusable components for repeated headers (such as menu bars). Thus, the number of possible test paths/cases that can be generated by DIG (and SUBWEB) is lower.

Despite such limitations, automated POs are competitive with manual POs in a subset of the considered applications. An interesting option available to testers could be the refinement of automatically generated POs, to achieve the same performance of manual POs at a lower development cost. Overall, our empirical results show that when high quality POs are available, our diversity based approach outperforms all other approaches both in terms of effectiveness (transition and branch coverage) and efficiency (rate at which coverage is reached).

Comparison with FeedEx. Te test generation approach implemented by DIG may seem similar to FeedEx [MFM13] conceptually in which event sequence (path) diversity and DOM structural diversity are considered for deciding how to explore a given web app. However, FeedEx is not a test generator but a crawler, hence a direct comparison is not possible. An interesting line for future work would be to develop a brand new tool that integrates FeedEx to retrieve the navigation model, and let Atusa or DIG to derive tests from it.

Chapter 5

Web Test Dependency Detection

E2E web test suites are prone to test dependencies due to the heterogeneous multi-tiered nature of modern web applications, which makes it difficult for developers to create isolated program states for each test case. In fact, in modern web applications the state is often distributed in multiple layers such as the database, the browser and third party services. Hence, it is difficult for developers to have the complete control over the state while developing the test cases.

Test dependencies are undesirable in the context of regression testing. In fact, many regression testing techniques, such as test prioritization [RUCH01], minimization [VSM18, RHVRH02], selection [HJL$^+$01] and parallelization [Kap16, BKMD15], assume test cases in the given test suite to be independent. Therefore, such techniques cannot be applied when dependencies among test cases are unknown, while new formulations of those problems can be devised if dependencies between tests are explicit (e.g. dependency-aware formulations of test parallelization [BKMD15] and test prioritization [M$^+$12] were proposed).

Manually detecting dependencies between tests is costly and infeasible in practice and the problem of automatically finding all possible dependencies in a test suite is *NP-complete* [ZJW$^+$14]. Therefore, researchers have focused their attention on devising automatic techniques that extract a subset of those dependencies in a timely manner. In particular, in this thesis, we are interested in finding those dependencies that need to be respected for the correct execution of the test cases in the given test suite.

Contribution. Heuristics to automatically extract dependencies in a test suite were developed for Java programs [ZJW$^+$14, BKMD15, GBZ18]. Essentially, test dependencies are extracted by detecting, statically or dynamically, read-after-write operations on the shared state between the tests, which is usually represented by static object fields in Java classes (for instance, the static variable x can be set, i.e. written, by t_1 and later read by t_2, which constitutes a read-after-write dependency over the variable x). Such analysis cannot be easily performed in E2E web test suites where the state is spread across multiple layers that involve multiple languages and technologies.

Table 5.1: Test cases for Phoenix Trello, numbered according to their test execution order.

Test	Name	Description
t_1	`addUserTest`	A new user account is created.
t_2	`loginUserTest`	The newly created user logs in to the application.
t_3	`addBoardTest`	The admin adds a board to his/her board page.
t_4	`shareBoardTest`	The admin shares his/her board with the newly created user.
t_5	`addMultipleBoardsTest`	The admin adds multiple boards to his/her board page.
t_6	`viewAllBoardsTest`	The admin accesses the menu to view all the boards in his/her board page.

In this thesis, we address the problem of automatically finding dependencies among E2E web tests. In a nutshell, our approach first extracts an approximated set of dependencies from the given test suite. It then filters potential false dependencies through natural language processing of test names. Finally, it validates all dependencies, and uses a novel recovery algorithm to ensure no true dependencies are missed in the final test dependency graph. Finally, we show that test dependency graphs extracted by our approach enable test parallelization and we measure the resulting execution speed-up factor.

5.1 Motivating Example

Ideally, running the tests in a test suite in any order should produce the same outcome [GBZ18]. This means tests should deterministically pass or fail *independently* from the order in which they are executed. Each test dynamically alters the state of the program under test in order to assert its expected behaviour, but in practice, some tests fail to undo their effects on the program's state after their execution, which can pollute any shared state [GSHM15] in tests executed subsequently.

In the web domain, testers perform end-to-end (E2E) testing of their applications [BWK05, FG99] by creating test cases using test automation tools such as Selenium WebDriver. Unlike unit testing, in which tests target specific class methods, web tests simulate E2E user scenarios, and therefore the program state that persists across test case executions might be left polluted, causing test failures if tests are reordered.

Motivating Example. Table 5.1 lists six E2E Selenium WebDriver tests for the Phoenix Trello web application (see Section 2.2.1.1).

Figure 5.1 shows dependencies between tests t_1 and t_2. The test case `addUserTest` navigates to the sign up form (line 3), fills the appropriate form with the details of the new user to create (lines 4–15) and it verifies that the user logged in is exactly the one just created (line 16). Finally, it logs out (line 17).

```
1  @Test
2  public void addUserTest() {
3     driver.findElement(By.xpath("//a[@href="/sign_up"]")).click();
4     driver.findElement(By.xpath("//input[@id="user_first_name"]")).clear();
5     driver.findElement(By.xpath("//input[@id="user_first_name"]")).sendKeys("foo");
6     driver.findElement(By.xpath("//input[@id="user_last_name"]")).clear();
7     driver.findElement(By.xpath("//input[@id="user_last_name"]")).sendKeys("bar");
8     driver.findElement(By.xpath("//input[@id="user_email"]")).clear();
9     driver.findElement(By.xpath("//input[@id="user_email"]")).sendKeys("foo@bar.com");
10    driver.findElement(By.xpath("//input[@id="user_password"]")).clear();
11    driver.findElement(By.xpath("//input[@id="user_password"]")).sendKeys("password");
12    driver.findElement(By.xpath("//input[@id="user_password_confirmation"]")).clear();
13    driver.findElement(By.xpath("//input[@id="user_password_confirmation"]"))
14       .sendKeys("password");
15    driver.findElement(By.xpath("//button[@type="submit"]")).click();
16    assertEquals("foo bar", ↩
          driver.findElement(By.xpath("//a[@class="current-user"]/span[2]")).getText());
17    driver.findElement(By.xpath("(//a[@href="#"])[2]")).click();
18 }
19 @Test
20 public void loginUserTest() {
21    driver.findElement(By.xpath("//input[@id="user_email"]")).clear();
22    driver.findElement(By.xpath("//input[@id="user_email"]")).sendKeys("foo@bar.com");
23    driver.findElement(By.xpath("//input[@id="user_password"]")).clear();
24    driver.findElement(By.xpath("//input[@id="user_password"]")).sendKeys("password");
25    driver.findElement(By.xpath("//button[@type="submit"]")).click();
26    assertEquals("foo bar", ↩
          driver.findElement(By.xpath("//a[@class="current-user"]/span[2]")).getText());
27    driver.findElement(By.xpath("(//a[@href="#"])[2]")).click();
28 }
```

Figure 5.1: Two dependent E2E web tests for the Phoenix Trello web application. Potential dependencies due to shared input data are highlighted.

The execution of `addUserTest` modifies the state of the web application, which is used by the subsequent test `loginUserTest` to login with the same credentials (`foo@bar.com`, `password`) created by `addUserTest` (lines 21–25). Thus, the shared input data (`foo@bar.com`, `password`, `foo`, `bar`) might reveal a potential dependency between the tests (see highlighted inputs in Figure 5.1).

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

In practice, however, testers re-use states created by preceding tests to avoid test redundancy, higher test maintenance cost and increased test execution time [LCRT16]. In doing so, they also enforce pre-defined test execution orders, which in turn inhibit utilizing test optimization techniques such as test prioritization [RUCH01].

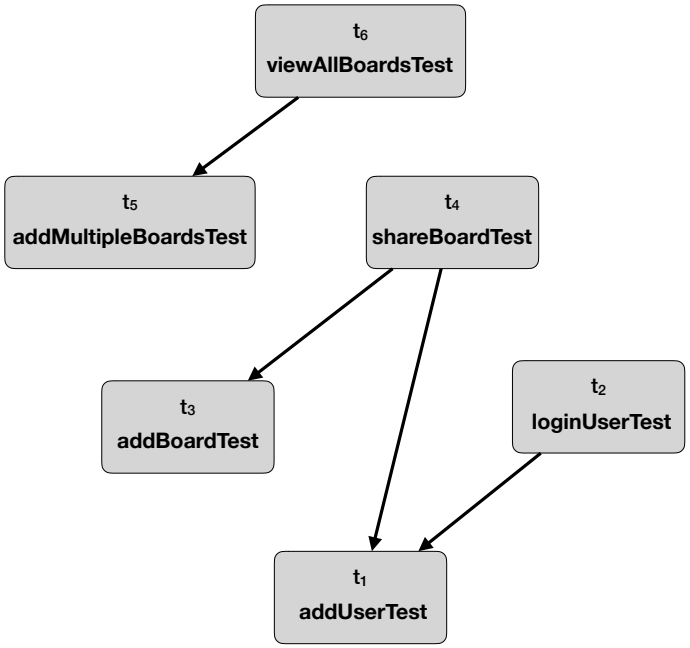


Figure 5.2: Test depencency graph for the test suite of Table 5.1. Solid edges represent manifest dependencies, namely, dependencies that result in a different outcome if unrespected.

Test Dependency Graph. Dependencies occurring between tests can be represented in a test dependency graph (TDG) [GBZ18], as presented in Section 2.2.2. Figure 5.2 illustrates the actual test dependency graph (TDG) for the test suite of Table 5.1

In order to be useful, TDG should contain all *manifest dependencies*, i.e., dependencies that do cause tests to fail if violated, while retaining the minimum number (or none) of *false dependencies*. One possible application of a TDG that contains only manifest dependencies is test suite parallelization. For instance, if we traverse the graph of Figure 5.2 and extract the subgraphs reachable from each node with zero out-degree (in our example there are three nodes, namely `addUserTest`, `addBoardTest` and `addMultipleBoardsTest`) we can identify subsets of tests that can be be executed in parallel with the others. In our example, three parallel test suites are possible: $\{ \langle t_1, t_2 \rangle, \langle t_1, t_3, t_4 \rangle, \langle t_5, t_6 \rangle \}$.

5.2 Approach

The goal of our approach is to automatically detect the occurrence of dependencies among web tests. Detecting dependencies in E2E web tests is particularly challenging due to the stack of programming languages and technologies involved in the construction of modern web applica-

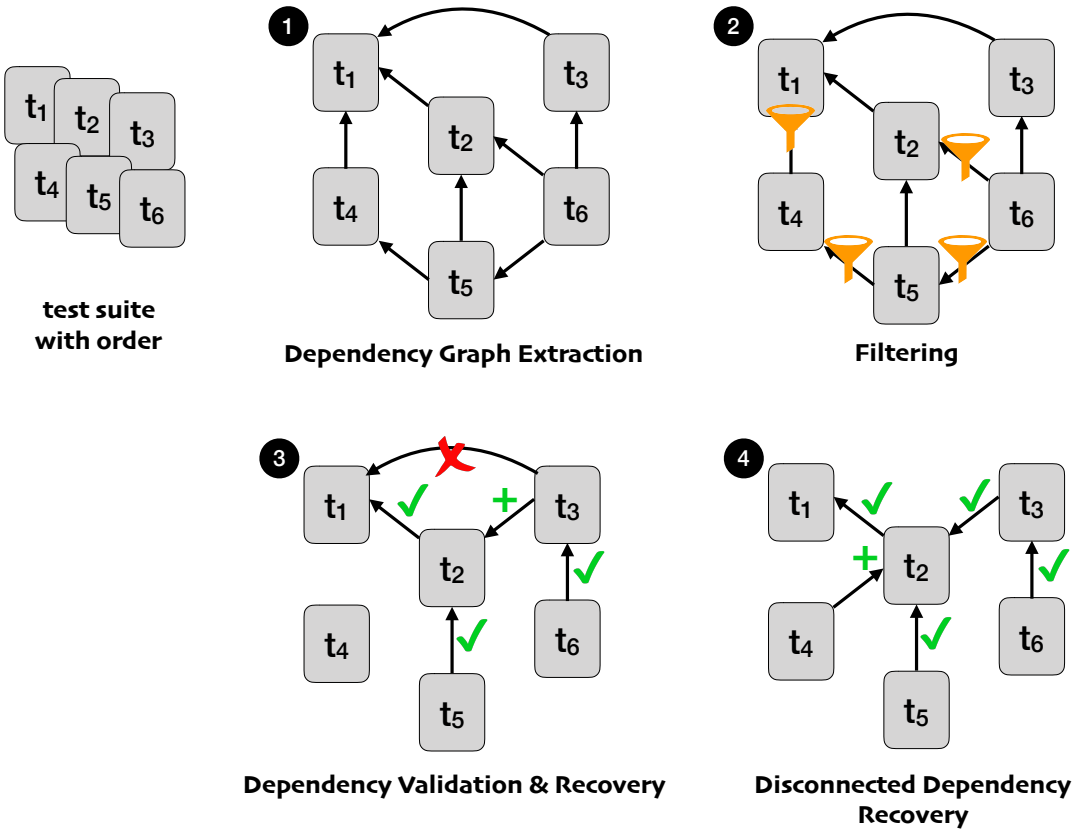


Figure 5.3: Our overall approach for web test dependency detection, validation and recovery.

tions, e.g., HTML, CSS, JavaScript on the client side; PHP, Java or JavaScript on the server-side, and back-end layers through Restful APIs or databases.

Our insight is that by analyzing the input data used in test cases (highlighted in Figure 5.1), we can obtain clues about potential test dependencies caused by shared polluted states. For example, the input string `password` used at line 11 in `addUserTest` is a *write operation* of persistent data. The same input string is used at line 24 in `loginUserTest` as a *read operation* of persistent data. We conjecture that such *read-after-write* connections on persistent data could indicate potential test dependencies. Given this insight, our approach focuses on read-after-write relationships on persistent data, defined as follows.

Definition 6 (Persistent Read-After-Write (PRAW) Dependency) *Two test cases t_1 and t_2 executed in this order in the original test suite are subject to a PRAW dependency if t_1 performs an operation that writes some information S_i into the persistent state of the web application and t_2 performs an operation that reads S_i from the persistent state of the web application.*

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

Figure 5.3 illustrates our overall approach, which requires a web test suite as input, along with a predefined test execution order. Overall, our approach ❶ computes an initial approximated test dependency graph, ❷ filters out potential false dependencies, ❸ dynamically validates all dependencies in the graph while recovering any missing dependencies, and, finally, ❹ handles missing dependencies affecting independent nodes possibly resulting from the previous validation step.

Next, we describe each step of our approach.

5.2.1 Dependency Graph Extraction

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

5.2.1.1 Original Order Graph Extraction

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

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

5.2.1.2 Sub-Use String Analysis Graph Extraction

Algorithm 4 describes our dependency graph extraction based on sub-use-chain relations. A sub-use relation consists of a submission (*sub*) of input data i and all the following *uses* of such submitted value i.

Algorithm 4: Sub-use string analysis graph extraction

Input : T_o: test suite in its original order o, $\mathcal{I}$: set of input-submitting actions
Output: TDG: test dependency graph with candidate dependencies to be validated

```
1  TDG ← ∅
2  T_a ← T_o                                              ▷ tests to analyze
3  foreach t in T_o do
4      S ← GETINPUTVALUES(t, I)                           ▷ S is the set of input values submitted by t
5      T_a ← T_a − {t}
6      foreach t_f in T_a do
7          U ← FINDUSEDVALUES(t_f) ∩ W                    ▷ U is the set of input values used by t_f
8          if U ≠ ∅ then
9              depToAdd ← (t_f --U--> t)                  ▷ candidate manifest dependency
10             TDG ← TDG ∪ { depToAdd }
11         end
12     end
13 end
```

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

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

The internal loop of Algorithm 4 (lines 6–12) searches for the input values indistinctly in any test action because in web tests there is no clear distinction between read and write statements. For instance, in Figure 5.1, the `sendKeys` action at line 9 is used to *write* persistent information into the application (e.g., in a database). However, the same `sendKeys` action, at line 22, identifies an action with a *read* connotation, because the string value `foo@bar.com` is used to verify that a specific persistent information (e.g. the email of a user) is present in the web application.

Let us consider the source code of the two test cases in Figure 5.1, namely `addUserTest` and `loginUserTest`. From the first test `addUserTest`, the algorithm extracts the set $\mathcal{S} = \{$`foo`, `bar`, `foo@bar.com`, `password`$\}$ because `sendKeys` is the only input-submitting action. Then, the string values in $\mathcal{S}$ are looked up in the subsequent test `loginUserTest`, producing the set of used values $\mathcal{U} = \{$`foo@bar.com`, `password`, `foo`, `bar`$\}$. Being $\mathcal{U}$ not empty, the algorithm creates a candidate test dependency between `loginUserTest` and `addUserTest`.

5.2.2 Filtering

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

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

5.2.2.1 Dependency-free String Value Filtering

We analyze the frequency of string input values used in the test suite to filter potentially false PRAW dependencies.

Let us consider the test dependency graph depicted in Figure 5.4, obtained by applying our sub-use string analysis graph extraction (Section 5.2.1.2) to the motivating example test suite (Section 5.1).

The set of dependencies $\{t_5 \rightarrow t_3, t_5 \rightarrow t_4, t_6 \rightarrow t_3\}$ represents instances of *false* candidate PRAW dependencies. The edges between these test cases are only due to the same login input data used by the tests—i.e., `john@phoenix-trello.com`, `12345678`—a default user created during the installation, for which no test case must be executed to create it.

Existing techniques [ZJW+14, GBZ18] refer to such cases as *dependency-free values*, i.e., if the test dependency graph includes dependencies that are shared across multiple (or all) test cases, these likely-false dependencies could be filtered out.

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

In our example of Figure 5.4, our approach computes the frequencies of all strings, presents it to the the tester, who, for instance, may decide to filter the dependencies due to the `john@phoenix-trello.com`, `12345678` pair of strings, hence removing $t_5 \rightarrow t_3$, $t_5 \rightarrow t_4$, $t_6 \rightarrow t_3$ (it can be noticed that the dependency $t_4 \rightarrow t_3$ is not filtered because such dependency is also due to the string value `myproject` which is not a dependency-free value).

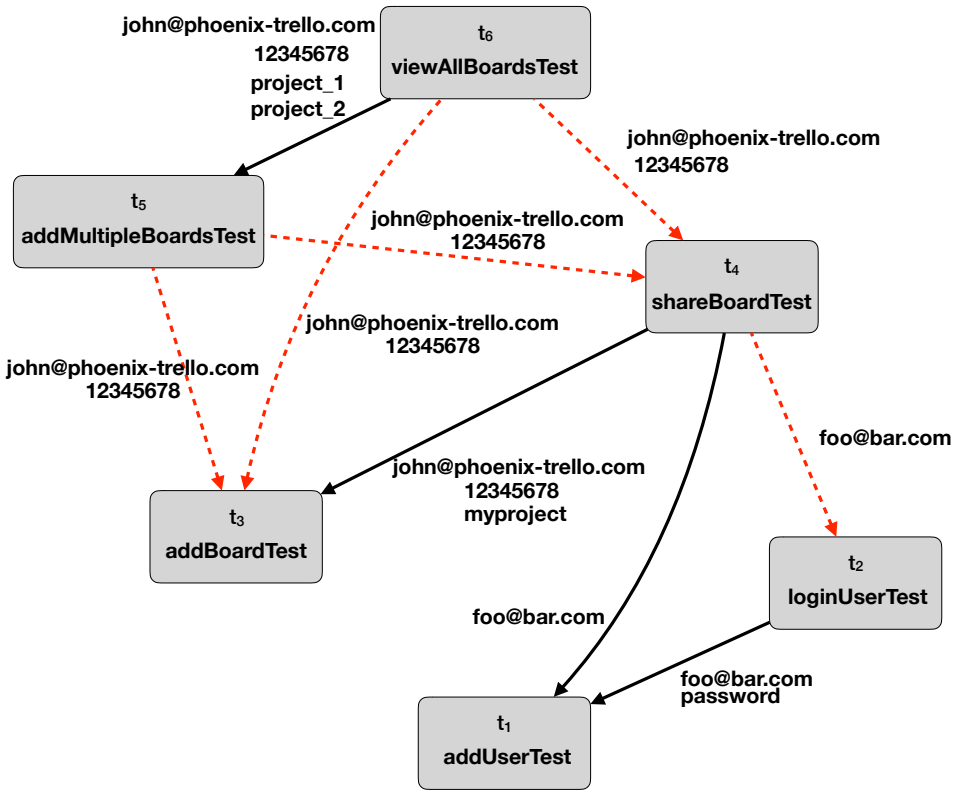


Figure 5.4: Dependency-free string-based PRAW filtering. Solid black edges represent manifest dependencies whereas dashed red edges represent false dependencies.

5.2.2.2 NLP-based Filtering

Developers often use descriptive patterns for test case names, which summarize the operations performed by each test. Giving a descriptive name to a test case has several advantages such as enhanced *readability* (i.e., it becomes easier to understand what behaviour is being tested) and *debugging* (i.e., when a test case fails, it is easier to identify the broken functionality). For instance, Google recommends test naming conventions [Goo19, Goo14] in which *unit tests* need to be named with the method being tested (a verb or a verb phrase, e.g., `pop`) and the application state in which the specific method is tested (e.g., `EmptyStack`). For *behaviour-based* tests such as E2E tests [Goo14], the guidelines propose a naming convention that includes the test scenario (e.g., `invalidLogin`) and the expected outcome (e.g., `lockOutUser`).

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

Our approach uses a *Part-Of-Speech tagger* (POS) to classify each token (i.e., word) in a tokenized test name as noun, verb, adjective, or adverb. In particular, our approach uses the verb from the test case name as the part of speech that conveys the nature of the test operation, and uses it to classify each test into *read* or *write* classes. Our approach relies on two groups of standardized R/W verbs, namely CRUD operations—*Create*, *Read*, *Update* and *Delete*) [CRU19]—in which the *Read* operation is pre-classified as read whereas the other three are pre-classified as write.

Our approach uses POS to extract the first verb from each test name and then computes the semantic similarity [PPM04] (specifically, the *WUP* metrics [WP94]) between the extracted verb and each verb in the pre-classified read/write groups. The similarity score quantifies how much two concepts are alike, based on information contained in the *is–a* hierarchy of *WordNet* [Mil95] (for example, an *automobile* might be considered more like a *boat* than a *tree*, if *automobile* and *boat* share *vehicle* as a common ancestor in an is–a hierarchy). After computing all similarity scores, our approach classifies the extracted verb to the group having the maximum similarity score. In case of ties (e.g., the verb has the same similarity score for both the read and write classes), or in case no verbs are found, our approach does not perform any assignment and the dependency is not filtered. Our classification of read/write verbs achieved a precision of 80% and a recall of 94% on our experimental subjects.

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

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

The dependency `shareBoardTest` → `loginUserTest` (Figure 5.4) is filtered because it is classified as WaR, with `login` being the read-classified verb. Conversely, the edge `loginUserTest` → `addUserTest` is retained since it is classified as RaW, being `login` and `add` the read/write verbs, respectively. The word `Test` is considered a *stop word* and removed before the NLP analysis starts.

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

RaR and WaR dependencies are filtered as described in the previous NLP Verb case. Differently, RaW and WaW dependencies are filtered only if the direct objects of two verbs appearing in two tests t_y and t_x are different. The intuition is that the two tests may perform actions on different

persistent entities of the web application, if the involved direct objects in the test names are different. For example, the WaW dependency `addListToBoardTest` → `shareBoardTest` is filtered because the two involved direct objects, `List` and `Board`, are different.

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

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

5.2.3 Dependency Validation and Recovery

Given a test dependency graph TDG, the overall dynamic dependency validation procedure works according to the iterative process proposed by Gambi et al. [GBZ18]. The approach executes the tests according to the original order to store the expected outcome. Next, it selects a target dependency according to a source-first strategy in which tests that are executed later in the original test suite are selected *first* (i.e., $t_3 \rightarrow t_2$ would be selected before $t_2 \rightarrow t_1$).

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

The dynamic validation procedure described above works correctly under the assumption that the initial TDG contains all manifest dependencies (as the original order graph Section 5.2.1.1). In our approach, the filtering techniques applied in the previous step may be not conservative. Therefore, differently from existing techniques [GBZ18], our approach features a dynamic dependency recovery mechanism that retrieves all *missing* dependencies. To the best of our knowledge, this is the first dependency validation algorithm that also includes dynamic dependency recovery.

Algorithm 5: Recovery algorithm

Input : T_o: test suite in its original order o
TDG: test dependency graph
targetDep: dependency selected for validation
expResults: results of executing T_o
execResults: results of a test schedule in which *targetDep* is inverted
Output : TDG: updated test dependency graph with missing dependencies recovered
Require: *expResults* $\neq$ *execResults*, i.e., *targetDep* is manifest

```
1  schedule ← COMPUTETESTSCHEDULEWITHNOINVERSION(TDG, targetDep)
2  execResults ← EXECUTETESTSCHEDULE(schedule)
3  failedTest ← GETFIRSTFAILEDTEST(expResults, execResults)
4  if failedTest ≠ null then
5      /* Failure due to a missing dependency; get all tests before failedTest. */
6      depCandidates ← GETDEPCANDIDATES(T_o, schedule)
7      foreach depCandidate ∈ depCandidates do
8          depToAdd ← ⟨ failedTest → depCandidate ⟩
9          TDG ← TDG ∪ {depToAdd}
10     end
11 end
```

Recovering Missing Dependencies. Algorithm 5 takes a partially-validated TDG. For each failing test schedule in which a target dependency is inverted, it checks whether the failure is due to a missing dependency in the dependency graph.

More specifically, Algorithm 5 takes the target dependency and computes a schedule in which the target dependency *is not* inverted (line 1). If the execution of such schedule complies with the expected outcome, our approach considers the test failure due to the dependency inversion and marks the dependency as a manifest. On the contrary, if one or more tests fail also in the schedule without inversion (line 4), our approach assumes that one or more dependencies are missing and need to be recovered.

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

Let us take as example Figure 5.5.A, in which t_4 has a *missing manifest dependency* on t_2, and t_3 does not modify the application state. According to the source-first strategy, the validation selects the dependency $t_4 \rightarrow t_3$. The schedule computed for such dependencies is $\langle t_4 \rangle$, in which t_4 fails because t_2 is not executed. Then, our algorithm starts retrieving the missing dependency by computing a schedule in which $t_4 \rightarrow t_3$ is not inverted, $\langle t_3, t_4 \rangle$, in which t_4 fails again for the

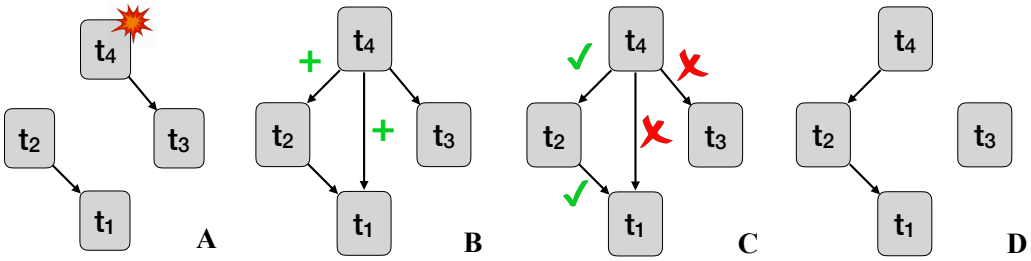


Figure 5.5: Recovery of missing dependencies. (A) $t_4 \rightarrow t_3$ is selected, t_4 fails because $t_4 \rightarrow t_2$ is missing. (B) recovery procedure adds candidate dependencies. (C) dependency validation. (D) final TDG.

same reason. The recovery procedure concludes that there is at least one missing dependency, and connects t_4 with both t_1 and t_2 (Figure 5.5.B), i.e., the only candidate manifest dependencies.

In Figure 5.5.C the dependency $t_4 \rightarrow t_3$ is selected again. This time, the computed schedule is $\langle t_1, t_2, t_4 \rangle$, in which none of the tests fail. Therefore, the dependency is marked as false and removed. The next selected dependency is $t_4 \rightarrow t_2$, for which the schedule $\langle t_1, t_4 \rangle$ is computed. The test t_4 fails because t_2 is not executed. To check if the failure is due to a missing dependency, our algorithm computes the test schedule $\langle t_1, t_2, t_4 \rangle$, in which none of the tests fail. Our algorithm concludes that $t_4 \rightarrow t_2$ is a manifest dependency and *recovers* it. The validation iterates over the other dependencies in the same way and outputs the final TDG (Figure 5.5.D), where the initially missing dependency ($t_4 \rightarrow t_1$) has been recovered.

5.2.4 Disconnected Dependency Recovery

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

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

Figure 5.6.A illustrates an example: the manifest dependency $t_4 \rightarrow t_1$ is missing in the initial dependency graph. Let us suppose that t_3 does not change the state of the application when executed, and therefore, its execution does not influence the execution of any other successive test in the original order. The algorithm selects first the dependency $t_4 \rightarrow t_3$, it produces the

[1] First test t_1 excluded

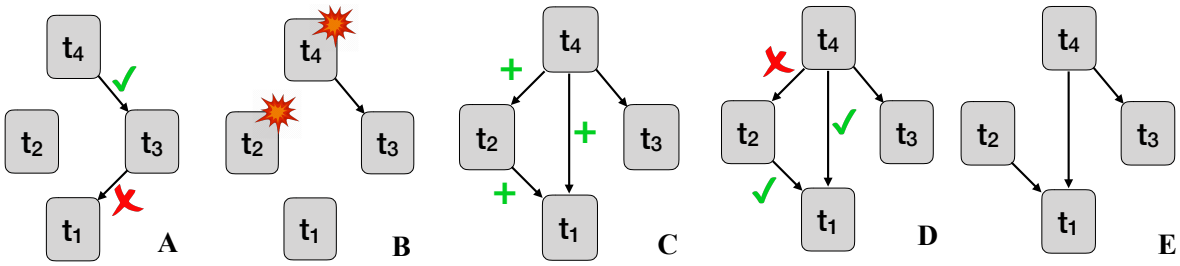


Figure 5.6: Disconnected dependency recovery. (A) dependencies are validated; $t_3 \rightarrow t_1$ shadows the missing dependency $t_4 \rightarrow t_1$. (B) t_4 and t_2 fail because of missing dependencies. (C) recovery procedure adds candidate dependencies. (D) dependencies are validated. (E) final TDG.

schedule $\langle t_4 \rangle$, in which the test fails since t_1 is not executed. Our approach checks if the failure is due to a missing dependency. When the dependency is *not inverted*, the computed schedule is $\langle t_1, t_3, t_4 \rangle$, in which none of the tests fail. Hence, our algorithm concludes that $t_4 \rightarrow t_3$ is a manifest dependency and no dependency recovery takes place. In the next step, the algorithm validates the dependency $t_3 \rightarrow t_1$, which is removed because t_3 can execute successfully without t_1. The dependency graph produced by dependency validation algorithm is illustrated in Figure 5.6.B. In the isolated subgraph $t_4 \rightarrow t_3$ the schedule $\langle t_3,t_4 \rangle$ results in a failure of t_4. Indeed, the dependency $t_4 \rightarrow t_1$ was not captured by the recovery algorithm because *the false dependency* $t_3 \rightarrow t_1$ *shadowed the absence of the manifest dependency* $t_4 \rightarrow t_1$.

Figure 5.6.A also illustrates how our approach handles isolated tests. In this example, t_2 is an isolated node. Let us suppose that t_2 has a manifest dependency on t_1 ($t_2 \rightarrow t_1$), which is missing in the initial TDG because it is either not captured, or because it is wrongly filtered out in the second step of our approach. Therefore, the validation step would produce the TDG shown in Figure 5.6.B, in which there is no chance to check whether t_2 executes successfully in isolation. In fact, t_2 is not part of any test schedule that can be generated from TDG, regardless of any possible dependency inversion. For this reason, a further recovery step is required once the validation is completed.

Algorithm 6 handles the recovery of missing dependencies within disconnected components. The algorithm retrieves all isolated nodes and zero out-degree nodes (line 3) and executes each of them in isolation (line 5). For each failing test, the algorithm connects it with all preceding tests according to the initial test suite order (line 8). Otherwise, if a test is not isolated and executes successfully (line 9), the algorithm takes all schedules that contain that test and execute them (lines 12–18). If a test in those schedules fails, the algorithm connects it with all the preceding ones (line 16).

Algorithm 6: Disconnected dependency recovery algorithm

Input : T_o: test suite in its original order o
TDG: test dependency graph
Output: TDG: updated test dependency graph with missing dependencies recovered

```
1  expResults ← EXECUTETESTSUITE(T_o)
2  /* Get isolated nodes and nodes with no outgoing edges. */
3  disconnectedTests ← GETDISCONNECTEDTESTS(TDG)
4  foreach disconnectedTest in disconnectedTests do
5      execResults ← EXECUTETESTINISOLATION(disconnectedTest)
6      failedTest ← GETFAILEDTEST(expResults, execResults)
7      if failedTest ≠ null then
8          TDG ← CONNECTWITHPRECEDINGTESTS(failedTest, TDG, T_o)
9      else if ISNOTISOLATED(disconnectedTest) then
10         /* Out-degree = 0; in-degree > 0. */
11         schedules ← COMPUTESCHEDULES(disconnectedTest, TDG)
12         foreach schedule ∈ schedules do
13             execResults ← EXEC(schedule)
14             failedTest ← GETFAILEDTEST(expResults, execResults)
15             if failedTest ≠ null then
16                 TDG ← CONNECTWITHPRECEDINGTESTS(failedTest,TDG, T_o)
17             end
18         end
19     end
20 end
```

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

Given the graph in Figure 5.6.B, Algorithm 6 executes t_2 in isolation, which fails, thus the dependency $t_2 \rightarrow t_1$ is added. Moreover, in Figure 5.6.B, there is only one schedule that involves t_3, namely $\langle t_3,t_4 \rangle$. In this schedule t_4 fails, hence our approach adds the dependencies $t_4 \rightarrow t_2$ and $t_4 \rightarrow t_1$ (Figure 5.6.C). Next, the added dependencies are validated (Figure 5.6.D), and the final graph is produced, where all initially missing manifest dependencies have been successfully recovered (Figure 5.6.E).

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

5.2.5 Implementation

We implemented our approach in a Java tool called TEDD (**Te**st **D**ependency **D**etector), which is available [2]. The tool supports Selenium WebDriver web test suites written in Java. TEDD expects as input the path to a test suite and performs the string analysis by parsing the source code of the tests by using *Spoon* (version 6.0.0) [PMP+15]. Our NLP module adopts algorithms available in the open-source library *CoreNLP* (version 3.9.2) [Cor19]. The output of TEDD is a list of manifest dependencies extracted from the final validated TDG.

5.3 Empirical Evaluation

We conducted an empirical study to answer the following research questions:

RQ_1 (effectiveness): How effective is TEDD at filtering false dependencies without missing dependencies to be recovered?

RQ_2 (performance): What is the overhead of running TEDD? What is the runtime saving achieved by TEDD with respect to validating complete test dependency graphs?

RQ_3 (parallel test execution): What is the execution time speed-up of the test suites parallelized from the test dependency graphs computed by TEDD?

[2] https://github.com/matteobiagiola/FSE19-submission-material-TEDD

Table 5.2: Subject systems and their test suites

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

5.3.1 Subject Systems

We selected six open-source web applications used in previous web testing research [LCRT13]. Each subject comes with one JUnit 4 test suite containing between 22-41 Selenium test cases; a JUnit test runner class specifies the tests order provided by the developer. Table 5.2 lists our subject systems, including their names, version, size in terms of lines of code, number of test cases, and the total number of lines of test code counted with `cloc` [3].

5.3.2 Procedure and Metrics

5.3.2.1 Procedure

We manually fixed any flakiness of the test cases of the subject test suites by adding delays where appropriate and we executed each test suite 30 times to ensure that identical outcomes are obtained across all executions. Moreover, at every schedule, we reset the state of the web application under test, both client and server side. This way each test dependency, checked in a schedule, is not affected by the execution of tests in previous schedules.

To form a *baseline* for comparison, we applied dependency validation to the dependency graph obtained from the original order of each test suite (Section 5.2.1.1). Essentially, our baseline approach represents the execution of Pradet's [GBZ18] heuristics on the TDGs obtained from the original test suites.

For each test suite, we ran different configurations of TEDD, by combining each admissible configuration of graph extraction and filtering technique. The first evaluated configuration is *String Analysis* (SA), in which the dependency graph is obtained through sub-use chain extraction (Section 5.2.1.2) and filtered from the dependency-free values (Section 5.2.2). Then, we evaluated three configurations in which we applied the three proposed NLP filters (NLP-Verb, NLP-Dobj, NLP-Noun) both to the graph from the original order as well as to the graph obtained with SA.

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

5.3.2.2 Metrics

To assess *effectiveness* (RQ_1), for each configuration we measured the number of *false dependencies* removed by TEDD as well as the number of manifest dependencies that are *missing* and

[3] `https://github.com/AlDanial/cloc`

need to be *recovered*. The number of false dependencies is obtained by subtracting the number of manifest dependencies retrieved at the end of the recovery step from the total number of dependencies in the initial graph.

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

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

5.3.3 Results

Effectiveness (RQ_1). For each configuration of TEDD, Table 5.3 (Effectiveness) shows the number of *extracted* dependencies starting from the initial test suite (Figure 5.3, step ❶) and the number of *filtered* dependencies (Figure 5.3, step ❷). It also reports information about the validation and recovery steps, specifically the number of *false* dependencies detected, the number of dependencies *recovered* and those recovered from the disconnected components. The final number (Column 8) shows the number of dependencies in the final TDGs, all of which are *manifest* PRAW dependencies.

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

TEDD does not ensure having minimal test dependency graphs, meaning that the final dependency graphs produced by TEDD can contain false positives (dependencies that are not manifest and could be filtered). Therefore, the number of manifest dependencies retrieved by each configuration is slightly different, between 32 and 34 (Column *Total PRAW*). However, these differences do not affect the executability of the schedules that respect the dependencies (see results for RQ_3).

Table 5.3: Effectiveness (RQ1), Performance (RQ2) and Parallelization (RQ3) average results across all subject test suites. Results for each subject test suite are available at `https://github.com/matteobiagiola/FSE19-submission-material-TEDD/blob/master/supplementary-material.pdf`

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

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

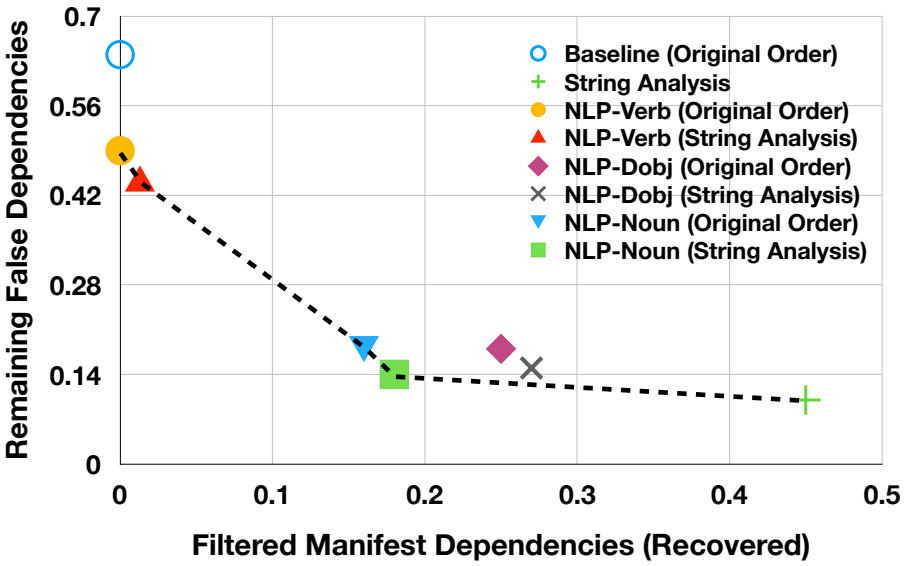


Figure 5.7: Pareto front

Figure 5.7 shows the Pareto front plotting the ratio between false and missing (or filtered manifest) dependencies, for each configuration. The Pareto front is used to represent the solutions of a multi-objective optimization problem [ES+03] where two or more conflicting objectives are optimized. In this work the two conflicting objectives to minimize are the number of remaining false dependencies after the filtering step and the number of filtered manifest dependencies (to be recovered) after the filtering step. In Figure 5.7 each point represents the average $\langle$*missing, false*$\rangle$ values across all subjects, normalized over the respective maximum values. This essentially shows the tradeoff between the false dependencies remaining after the filtering step and the missing dependencies to be recovered. The blacked dotted line in Figure 5.7 is the Pareto front which is a set of *non-dominated solutions* to the optimization problem. On the other hand, a solution x_1 is referred to as *dominated* by another solution x_2 if, and only if, x_2 is equally good or better than x_1 with respect to all objectives. For instance, the solution/configuration NLP-Verb (Original Order) (yellow circle in Figure 5.7) dominates the solution/configuration Baseline (Original Order) (empty blue circle in Figure 5.7) because NLP-Verb (Original Order) has a lower value (i.e. it is better) than the Baseline (Original Order) configuration with respect to the number of false dependencies objective.

From the analysis of the Pareto front, we can see that the non-dominated configurations are those based on NLP-Verb, NLP-Noun and String Analysis (SA). The baseline approach (Baseline) has the highest number of false dependencies (536 on average) and no missing dependencies. On the contrary, SA filters many dependencies (393 on average) but has the highest number of missing/recovered dependencies (11 on average). Interestingly, NLP-Verb (Original Order) does not

miss any manifest dependency but has more false dependencies compared to the other NLP-based configurations. Configurations NLP-Dobj (both SA and Original Order) and NLP-Noun (both SA and Original Order) are comparable regarding the number of false dependencies remaining after filtering. However, NLP-Noun (SA and Original Order) needs to recover substantially less manifest dependencies. Indeed, NLP-Noun dominates NLP-Dobj, while both NLP-Noun (SA and Original Order) configurations are on the non-dominated front, being both optimally placed in the lower-left quadrant of the Pareto plot.

Performance (RQ_2). Table 5.3 (Performance) reports the average runtime, in minutes, of each step of TEDD across all configurations. The most expensive step of TEDD is validation, especially for what concerns validating the connected part of the graph (Column 12), whereas dependency graph extraction and filtering (Columns 10 and 11) have negligible costs (under one minute on average in all cases). The cost of disconnected components recovery (Column 13) is generally low, ranging from nearly one minute for NLP-Verb to maximum nine minutes for NLP-Dobj (3.8 minutes on average).

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

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

Parallel Test Execution (RQ_3). Column 15 (*schedules*) reports the average number of test schedules obtained from the final TDGs. Isolated nodes in the dependency graphs are counted as (single-test) schedules. Columns 16 and 17 report the relative speed-up of the parallelizable test suites considering the longest test execution schedule (worst-case) and the average case.

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

Overall, all techniques achieve similar speed-up scores, around 2$\times$ in the worst case and 7$\times$ on average. This is expected since the final TDGs are similar across configurations (see total number of manifest dependencies in Table 5.3). However, results differ across applications. Table 5.4 presents the results for the NLP-Noun (SA) configuration. Column *runtime original* reports the execution time of the original test suite in seconds. *Collabtive* has the slowest test suite (almost 5 minutes) whereas *Addressbook* has the fastest (40 seconds). The highest speed-up in the worst-case occurs for *Claroline*, where the longest test execution schedule test suite is 2.7$\times$ faster. *MantisBT* exhibits the highest speed-up in the average case (12.3$\times$), but the lowest speed-up in the worst-case (1.4$\times$) due to a single slow-executing schedule (133 s) with respect to the average

Table 5.4: Parallelization results for NLP-Noun (SA)

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

* *average*

runtime (15 s). The lowest speed-up in the average case occurs for *MRBS*, but it remains still high (4.2×).

5.3.4 Threats to Validity

Using a limited number of test suites in our evaluation poses an *external validity* threat. Although more subject test suites are needed to fully assess the generalizability of our results, we have chosen six subject apps used in previous web testing research, pertaining to different domains, for which test suites were developed by a human web tester. Threats to *internal validity* come from confounding factors of our experiments, such as test flakiness. To cope with possible flakiness of the test cases, we manually fixed any flaky test by adding delays where appropriate and we ran each test suite 30 times to ensure having identical results on all executions. With respect to reproducibility of our results, the source code of TEDD and the *Docker* containers for all subject systems are available online [4], making the evaluation repeatable and our results reproducible.

[4] `https://github.com/matteobiagiola/FSE19-submission-material-TEDD`

5.3.5 Discussion

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

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

Test Smells. Our test dependency graph can also be utilized for other test analysis activities such as detecting *poorly designed* or *obsolete* tests (i.e., test smells [vDMBK02]). For instance, in *PPMA*, the test `checkEntryTagsRemoved` executes after `addEntryTags` and `removeEntryTags` tests. By analyzing the TDG produced by TEDD for this test suite, we noticed that `checkEntryTagsRemoved` executes properly also when no tags have been created yet (i.e., `checkEntryTagsRemoved` is *isolated* in the TDG). This means that the test `checkEntryTagsRemoved` is obsolete because it is subsumed by the previous `removeEntryTags` test. Therefore, it can be safely removed with no impact on the functional coverage or assertion coverage of the test suite.

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

Chapter 6

Dependency Aware Test Case Generation

Crawlers are appealing tools for testers as they can effectively and rapidly explore the state space of a web application. However, deriving executable functional test cases from a sequence of crawled pages and events is not trivial. First, such sequence is typically very long and it would be inefficient to execute it directly within a single test case. Therefore, the sequence must be *segmented* into meaningful sub-sequences, according to some criteria, in order to isolate the different functionalities exercised by the crawler into separate test cases. Second, the obtained sub-sequences may have *dependencies* on web application states created by previous sub-sequences [GSHM15, ZJW+14, LZE15], which must be resolved. Third, since crawlers are designed to perform continuous, repeated explorations with different randomly generated inputs, the segmented sequences may also be *redundant*, which increases the test suite runtime, without benefiting the overall coverage of the web application functionalities.

Existing works [MvDL12, MFM13, YMZ15] use crawlers for test generation quite differently. Indeed, crawlers are used to navigate the WAUT (Web Application Under Test) and create a navigation model which is then used for test generation purposes. Given some model based adequacy criterion, such as transition coverage, tools like Atusa [MvDL12] derive a set of paths from the navigational model. Such paths represent abstract test cases that can be turned into concrete, executable test cases by supplying proper input values. The main limitation of such model-based test generation techniques is that they have to cope with the *feasibility problem*, i.e., finding proper paths in the navigational model along with associated input values, such that, upon execution of the web application under test, the desired navigation is taken (see Section 4.3 and Section 2.3.1 for the definition of path feasibility). Determining if a path is feasible is in general undecidable and in practice a very challenging problem. Infeasibility might be due to dependencies on previous states that the selected path cannot reproduce (i.e. state dependencies) or on the impossibility to generate input values satisfying the path constraints (see Section 4.1).

Contribution. In this chapter, we propose a novel approach to web test generation that combines *crawling*, *test dependency detection*, and *test minimization*. Our approach, implemented in a tool called DANTE (**D**ependency-**A**ware Crawli**n**g-Based Web **T**est G**e**nerator), takes as input the raw segmented crawling trace produced by a crawler, computes and validates the test dependencies between each sub-sequence, ensuring that any resulting test suite (sub-list of segmented sequences) that respects them will not result in any test breakage. Last, DANTE uses the test dependencies as a set of constraints for detecting and eliminating redundant test cases using a SAT solver.

6.1 Motivating Example

In Section 2.3.2 we described the functioning of the web crawler Crawljax [MvDR12]. In this section we recall the exploration strategy and state abstraction function concepts and examine them in detail. Then, we present a motivating example to show the main limitations of crawlers when they are used to generate web tests.

Three are the important characteristics of a web crawler that impact test generation, namely *exploration strategy*, *state abstraction function* and *sequence segmentation*.

Exploration Strategy. The crawler explores the web application according to a graph visit algorithm, such as breadth-first or depth-first visit [Mes15, TRM14]. The default option in Crawljax is the depth-first visit: on each state, one clickable is selected, and fired an event upon. The order in which candidate elements are selected within a state can be changed. For instance, FeedEx [MFM13] selects the clickable element to consider based on a linear combination of factors aimed at maximizing the diversity of the exploration.

Another aspect that impacts the crawler's exploration is the selection of the same clickable element multiple times within different states. For instance, let us consider a web application having a navigation bar with menu items which are displayed in all possible states. The tester can decide whether Crawljax should consider each menu item *only once* during exploration, or, differently, whether it should consider them *multiple times* in different states. The rationale for choosing the former option is to make the exploration faster, assuming that all menu items, when fired upon, always bring the web application to the same state deterministically. However, this assumption might not always be true, especially for modern web applications such as single page web applications. Therefore, the latter option may ensure a more thorough exploration.

State Abstraction Function. To avoid redundancies, states that are identical or similar to previously encountered states should be discarded. The problem of detecting already visited states is delegated to the state abstraction function, which is a function (which returns a value between 0.0 and 1.0) that decides if a new state is found after an event is fired. The default state abstraction function integrated within Crawljax compares the equality of the string representation of the

DOM of each web page, which ensures fast comparisons, and therefore, more exploration capabilities. In such particular case, indeed, the state abstraction function returns 0.0 if the two DOM strings are the same and 1.0 if they are different. However, other state abstraction functions have been proposed, e.g., comparing the DOM tree by the tree edit distance [MFM13]. In this case a threshold τ, between 0.0 and 1.0, has to be defined to decide if the two states are different (i.e. the value returned by the state abstraction function is greater than τ) or not.

Sequence Segmentation. Given a web application's URL, an exploration strategy, and a state abstraction function, the crawler performs an exploration of the web application state space and returns a (possibly long) sequence of visited pages. Then, it is possible to segment such sequence into shorter sub-sequences that can be used as test cases, which replay the actions of the crawler on the web application. On the contrary, replaying the entire sequence at once would require as much execution time as the crawling phase, if used as a single test case. One approach is to segment the crawling sequence whenever no more candidate elements are present in a given state, or no new DOM states are discovered in a given crawl path. In this case, the sequence is *ended* (i.e., segmented) in that state, and the crawler continues its exploration from the first unvisited state, or from the start state (usually, the index page).

6.1.1 Crawling Trace Based Test Generation and its Limitations

The list of segmented sub-sequences retrieved by a crawler represents candidate test cases. However, two main problems may occur and need to be addressed.

First, after segmentation, the individual test cases may be *dependent* on each other. Test dependencies are due to the web application state being modified by actions performed by the crawler in previously executed test cases.

Second, test *redundancy* may appear as a consequence of the length of the navigation performed by the crawler. On the one hand, long navigations are desirable, because they have more chances to explore the web application in depth. However, long navigations tend also to include many segments that are equivalent (according to some chosen adequacy criterion, such as code/model coverage). Other factors affecting the degree of test redundancy are the crawler's state abstraction function and the strategy used to select DOM elements within a state. For what concerns state abstraction, when the state abstraction function is too coarse-grained, many parts of the web application would be unexplored because many states would be considered the same. Conversely, if the state abstraction function is too permissive, many similar states would be considered different, leading to many redundant test cases. For what concerns DOM element selection, when the crawler considers candidate elements *multiple times*, it can explore the web application more deeply, at the cost of potentially increasing the redundancy of the tests.

Figure 6.1 (left) shows the web application *Phoenix Trello* whose index page contains a navigation bar to navigate to the index page (the logo *Phoenix Trello*), to the *Login* page (button *Sign*

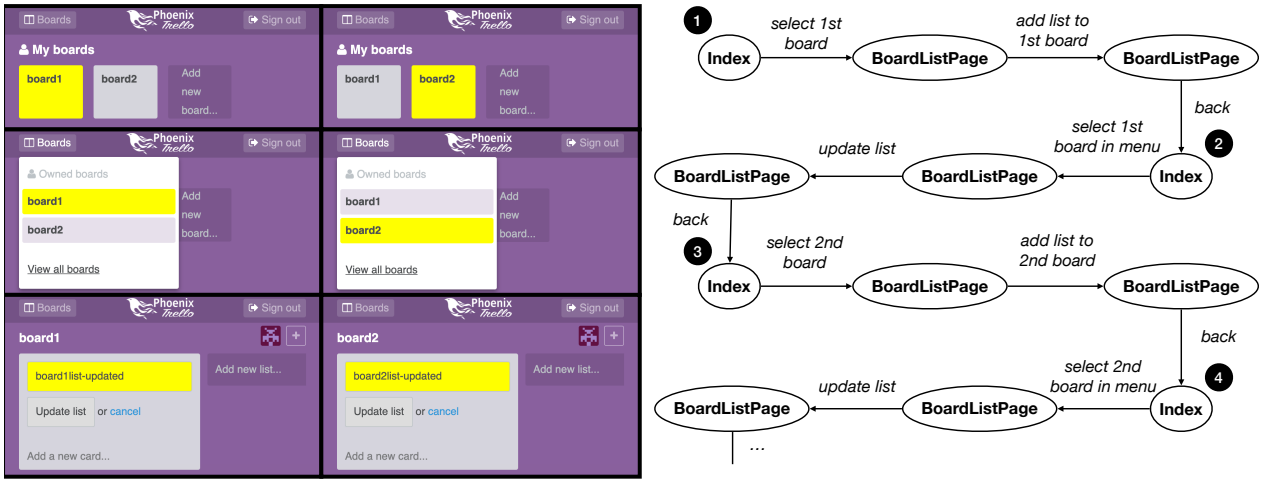


Figure 6.1: Phoenix Trello example and generated crawling trace containing dependent and redundant test cases.

Out), and to the *List* page of each board (button *boards*). When the *boards* button is clicked a dropdown menu with all the existing boards in the web applications is shown (second screenshot on the left hand side of Figure 6.1). Clicking on the name of a board brings the web application to the page that shows the lists created in each board (namely the *BoardListPage*). Once a list is created it is possible to update its name (third screenshot on the left hand side of Figure 6.1).

Crawling. Figure 6.1 on the right hand side shows a possible exploration performed by the crawler on the Phoenix Trello web application that results in a long sequence of visited states. For simplicity, suppose the crawler adopts the default depth-first crawling exploration strategy (clicking the same web elements only once) and a state abstraction function based on DOM string equality.

In the represented sequence, the crawler selected the first board (*board1*) by clicking on it in the index page. Then, it created a list for that board and navigated back to the index page. Then, the crawler accessed the menu in the index page, clicked on the *board1* link and updated the name of the list, before returning to the index page. The same crawling path was repeated for the other board (*board2*); the list name for the second board was also updated.

Segmentation. Suppose that in our example, the crawler segments the crawling trace whenever it reaches the first visited page (e.g., *Index*). In this case, four sub-sequences are generated, as shown in Figure 6.1 (the first state of each new sub-sequence is marked by an incremented number). For instance, the first sub-sequence ❶ ends after having added the list to the first board, whereas the second sub-sequence ❷ ends after having updated the list name for the first board. The other sub-sequences are segmented similarly.

Dependent Tests. After segmentation, the resulting test cases may be dependent. Test dependencies are due to the application state being modified by the actions the crawler performed in previously executed tests. For instance, sub-sequence ❷ (i.e., updating the name of the list for the first board) depends on sub-sequence ❶ (i.e., creating the list for the first board). Similarly, sub-sequence ❹ depends on sub-sequence ❸.

If we execute each of those tests in *isolation*, the first test ❶ executes correctly, while the second one ❷ breaks. Indeed, the initial application state does not contain the list that the second test needs to update, since that item is created by the first test.

On the contrary, if tests obtained after segmentation are executed in the same order in which they were crawled, no breakage occurs (i.e., by running in sequence tests ❶–❹). If we want to remove redundant test cases, the dependencies between tests should be known and should be taken into account [ZJW+14, LZE15, GBZ18, BK14, RUCH01, GEM15, VSM18].

Redundant Tests. The test cases shown in Figure 6.1 may be redundant for a given test adequacy criterion, such as client-side code coverage. The first two sub-sequences ❶–❷ cover a test scenario in which a board is selected and the name of its list updated. Sub-sequences ❸–❹ essentially repeat the same scenario, with different data. If such boards are retrieved by the server-side of the web application, the overall client-side code coverage would not change when executing these last two sub-sequences. Therefore, if the adequacy criterion is client-side code coverage, one of these two sub-sequences could be safely removed without affecting the adequacy of the test suite.

6.2 Approach

Figure 6.2 illustrates the overall approach, which takes as input a segmented crawling sequence retrieved by a crawler when executed on a given web application. Test dependency analysis is used to retrieve and validate the dependencies between each sub-sequence (*Test Dependency Analysis*), which are subsequently used as a set of constraints within a SAT solver to eliminate redundant test cases (*SAT solver-based Test Minimization*).

Test dependency analysis is however a computationally expensive task, because the total number of possible dependencies is quadratic in the number of test cases and research has shown that automatically identifying the subset of true dependencies may require exponential analysis time [BKMD15, GBZ18, ted19]. In fact, in the absence of any preliminary filtering, each test case in the original sequence must be assumed as possibly dependent on all its predecessors, resulting in $n(n-1)/2$ (with n the number of test cases) candidate dependencies to be validated by dependency analysis, of which only a small subset may actually result in true dependencies.

To reduce the cost of dependency analysis, in this work we propose and evaluate *two alternative filtering heuristics* that can reduce the initial number of dependencies to be validated.

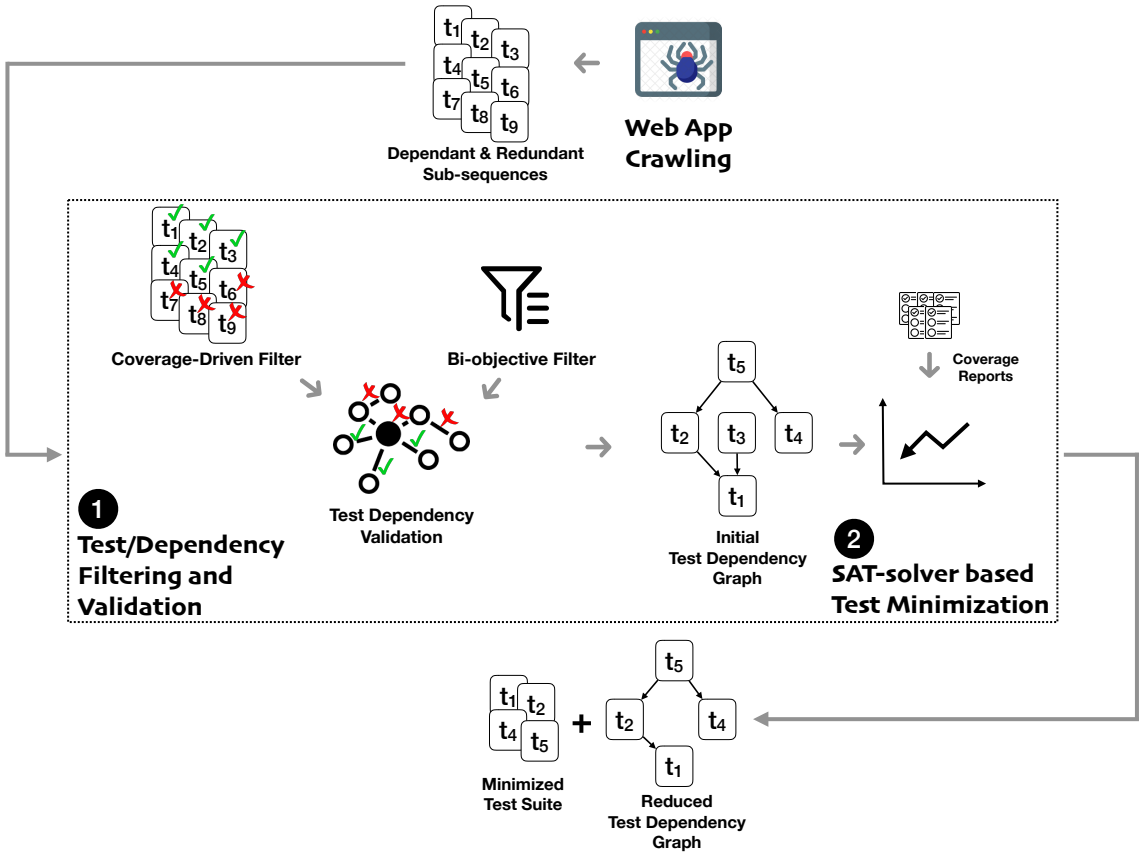


Figure 6.2: High-level overview of our approach for dependency-aware web test generation and minimization

The first filtering heuristics is based on pre-selecting only the tests that contribute to web application coverage (*coverage-driven filter*), while eliminating the unnecessary ones, and making sure that dependents of the selected tests are included.

The second filtering heuristics is based on bi-objective optimization (*bi-objective filter*), where the two objectives being minimized are (1) the number of dependencies kept for successive dependency validation, and (2) the estimated cost of recovery of the incorrectly filtered dependencies.

In the second phase, test suite minimization aims at eliminating all redundant test cases, i.e., tests that do not contribute to coverage and are not needed due to some dependency. This second phase coverage based minimization is still needed to eliminate all redundant tests even if the first filtering heuristics (based on coverage as well) is applied. The reason is that spurious redundant tests can be included by the non-optimal (i.e. non-minimal) coverage-driven filter.

The output of our approach is therefore a *minimized* test suite in which (1) no redundant test case are present, and (2) all test schedules that respect the dependencies can execute independently and without errors.

6.2.1 Test Dependency Analysis

For dependency retrieval and validation we use our existing tool named TEDD (see chapter 5), which automatically detects the occurrence of dependencies among web tests. From a given test suite, TEDD retrieves an initial set of dependencies to be validated, which may include both spurious dependencies to be removed and missing dependencies to be recovered. The output of TEDD is a test dependency graph such that all test schedules that respect its dependencies execute correctly.

We proposed TEDD and evaluated it for validating dependencies in human-written tests, for which appropriate filtering techniques, based on natural language processing of the test case identifiers (i.e. test case names), have been evaluated. In this work, we target automatically generated test suites, which do not include meaningful identifiers. Our preliminary experimental results have shown that TEDD does not terminate within a timeout of 48 hours (2 days), when applied to the complete (quadratic), unfiltered test dependency graph. Hence, we designed two novel filtering heuristics, aiming at making TEDD applicable to automatically generated test cases.

6.2.1.1 Coverage-Driven Filter

The first proposed filter consists of retaining only the tests that contribute to the coverage of the web application, along with their dependent tests. The filter is independent from the coverage criterion (e.g., any of model/transition or code/branch coverage could be adopted). We hereafter refer to *element* as the unit of coverage specific to the selected coverage criterion (e.g., a transition for transition coverage, or a branch for branch coverage).

The filtering is performed as follows. The initial test sequence is executed in the order retrieved by the crawler, to gather the needed coverage information. A greedy algorithm starts by selecting the test that achieves maximum coverage. Then, it repeatedly adds the test that covers more additional elements, with respect to the coverage achieved by the already selected test cases, until the final coverage of the whole test suite is reached. Then, the so obtained filtered test suite is executed. If no breakages occur, our approach continues with the dependency analysis by TEDD. On the contrary, if tests break because some other test needs to be executed first, an automated fixing procedure is triggered, which we detail next.

Fixing Missing Test Dependencies.

Algorithm 7: Test suite fixing algorithm

Input : T_s: test suite with coverage driven selected test cases
T_o: test suite in its original order o
Output: T_s: updated test suite with test dependencies fixed

```
1  brokenTest ← EXECUTETESTSUITE(T_s)
2  if brokenTest = null then
3  |   return T_s
4  end
5  windowLength ← 1
6  while true do
7  |   preconditions ← COMPUTEPRECONDITIONS(T_o, T_s, brokenTest)
8  |   newBrokenTest ← null
9  |   i ← windowLength
10 |   while i < |preconditions| do
11 |   |   preconditionsToAdd ← SUBSET(preconditions, i, windowLength)
12 |   |   newBrokenTest ← ADDANDEXECUTE(preconditionsToAdd, T_s)
13 |   |   if newBrokenTest = null then
14 |   |   |   return T_s
15 |   |   end
16 |   |   if newBrokenTest ≠ brokenTest ∧ ORDER(newBrokenTest) > ORDER(brokenTest) then
17 |   |   |   break
18 |   |   end
19 |   |   T_s ← REMOVEPRECONDITIONS(preconditionsToAdd, T_s)
20 |   |   i ← i + windowLength
21 |   end
22 |   if newBrokenTest = brokenTest then
23 |   |   windowLength ← windowLength + 1
24 |   else
25 |   |   brokenTest ← newBrokenTest
26 |   end
27 end
```

Algorithm 7 shows our automated procedure for fixing missing dependencies. The algorithm takes as input the selected test suite T_s and the original test suite T_o as generated by the crawler.

The EXECUTETESTSUITE procedure executes T_s and, if no breakages are detected (line 3), the algorithm terminates. On the contrary, the first test case that breaks is returned (*brokenTest*), and considered for dependency fixing.

The algorithm computes the preconditions of *brokenTest* (line 7) by considering all tests in the original test suite T_o that are placed before it, and that are *not* yet included in T_s. The loop (lines 6–27) adds one precondition at a time (initially, *windowLength* $= 1$), and checks if the

test suite with the added precondition executes correctly (line 12). If so (*newBrokenTest* = *null*), the algorithm terminates. Otherwise, the previously added preconditions are removed from T_s (line 19) and the loop (lines 6–27) continues. The loop is interrupted also when *newBrokenTest* follows (hence, it has to replace) *brokenTest* (lines 16–17). The ORDER function computes the index of the test case given as input in the original test suite T_o. Such index defines an order relation between test cases that corresponds to the execution order given by crawler segmentation (Section 6.1.1). If *newBrokenTest* is equal to *brokenTest* (lines 22–23), the size of the window is increased.

Let us consider the test suite $T = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9\}$. Suppose that the coverage-driven filter selects t_1 and t_5, hence $T_s = \{t_1, t_5\}$. Now let us suppose that t_5 depends only on t_2 and on t_4 (indicated as $t_5 \rightarrow t_2$ and $t_5 \rightarrow t_4$). Algorithm 7 tries to execute T_s but t_5 breaks. The preconditions of t_5 are $\{t_2, t_3, t_4\}$. The algorithm tries to add them one at a time, but each one in isolation is not sufficient to fix t_5. Then, at the end of the first iteration, the window size is increased to 2. The algorithm adds $\{t_2, t_3\}$ first and then $\{t_3, t_4\}$ but none of the two sets of tests fixes t_5, therefore they are both removed from T_s. Finally, the window length is increased to 3 and the tests $\{t_2, t_3, t_4\}$ are added. The broken test t_5 passes and $T_s = \{t_1, t_2, t_3, t_4, t_5\}$ is the fixed filtered test suite. It can be noticed that T_s contains one spurious dependency, t_3, to be removed by TEDD during dependency analysis. Such a spurious dependency is a consequence of the adopted heuristics, which adds all test cases inside the current window, rather than taking all the possible subsets of the current window (the latter would incur an exponential computational cost). Moreover, the test suite minimization phase makes sure that the redundant test t_3 is eliminated from the final test suite.

6.2.1.2 Bi-objective Filter

While the first proposed filter aims to reduce the number of initial tests (thus making the initial test dependency graph smaller), the second proposed filtering aims at reducing the number of dependencies from the complete (quadratic) graph. The general idea is to execute TEDD on a filtered graph that contains as few dependencies as possible, while at the same time minimizing the estimated (worst case) recovery cost that TEDD may incur when identifying and re-introducing missing dependencies.

Let $T = \{t_1, \ldots, t_n\}$ be the set of all tests and let $D = \{d_{11}, \ldots, d_{nn}\}$ be the set of all dependencies associated with the original crawl order. Each dependency d_{ij} is defined as:

$$d_{ij} = \begin{cases} 1, & \text{if } i > j. \\ 0 & \text{otherwise.} \end{cases}$$

Let $S = \{s_{11}, \ldots, s_{nn}\}$ be a filtering matrix over D (s_{ij} is 0 if the dependency is filtered; 1 otherwise; moreover, $s_{ij} = 0$ for all $j \geq i$). For each test t_i with $2 \leq i \leq n$, let us consider all

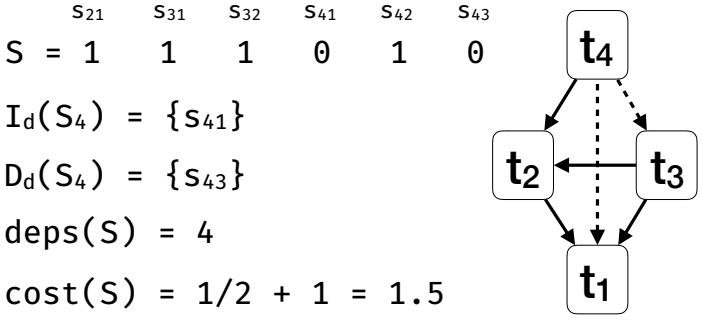


Figure 6.3: Bi-objective filtering example

outgoing dependencies that are filtered in S (i.e. $s_{ij} = 0$, while $d_{ij} = 1$). Let $D_d(S_i)$ be the set of filtered dependencies of node i which are *direct*, i.e. there exists a path in S that could connect such pair of tests. Let $I_d(S_i)$ be the set of filtered dependencies of node i which are *indirect*, i.e., no other path in S exists between such pair of tests. By construction, these two sets are disjoint ($D_d(S_i) \cap I_d(S_i) = \emptyset$). Correspondingly, we define the following two objective functions:

$$\textit{minimize} \quad \textit{deps}(S) = \sum_{s_{ij} \in S} s_{ij} \tag{6.1}$$

$$\textit{minimize} \quad \textit{cost}(S) = \sum_{i=2}^{n} \left(|D_d(S_i)| + \frac{1}{2}|I_d(S_i)| \right) \tag{6.2}$$

Equation 6.1 simply counts the number of unfiltered dependencies in S; such sum has to be minimized. Equation 6.2 estimates the cost of recovery of each dependency that is filtered in S. In the worst case, all filtered direct ($D_d(S_i)$) and indirect ($I_d(S_i)$) dependencies must be recovered. However, the second contribution to the cost is divided by two, because each indirect filtered dependency might be already included in the graph through other dependencies, provided the validity of each dependency in the indirect path has already been confirmed by the dependency analysis (i.e., TEDD).

We solve this minimization problem by means of bi-objective optimization [ES+03]. The optimal solution is a Pareto front with two dimensions (*deps* and *cost*), populated by the non-dominated solutions (i.e., filtering matrices) discovered by the algorithm (see Section 5.3.3 for an example of Pareto front). To obtain one single filtering matrix from the Pareto front of solutions, we compute the derivative on each point in the Pareto front and choose the point with highest derivative (i.e., highest gain on both *deps* and *cost*).

Figure 6.3 shows an example of how the two objective functions are computed. S is the specific dependency filter that is represented by the dependency graph on the right hand side of the figure,

with dashed arrows representing filtered dependencies. There are only two filtered dependencies in S, hence $deps(S) = 4$. The first one s_{41}, which corresponds to $t_4 \rightarrow t_1$, is a filtered indirect dependency, because there is a path between t_4 and t_1 that passes through t_2. If $t_4 \rightarrow t_1$ were true, it would have to be recovered only if $t_4 \rightarrow t_2$ or $t_2 \rightarrow t_1$ are deemed as invalid during dependency validation. This is why the cost of recovering an indirect dependency has a lower (half) weight than the cost of recovering a direct dependency, such as s_{43}.

6.2.2 SAT solver-based Test Minimization

Our approach adopts a SAT solver [ACA12] to find optimal solutions to the test suite minimization problem. We encode the test dependencies as a set of pseudo-boolean constraints that are translated to a SAT instance. Then, the SAT instance is solved using a SAT solver.

The problem formalization is as follows. Let us introduce n boolean variables $t_i \in \{0, 1\}$, one for each test case in T. If $t_i = 1$ then the corresponding test case is included in the solution, otherwise ($t_i = 0$) the test case is excluded. Let $C = \{c_1, \ldots, c_n\}$ be the set of execution costs of running each test (with $c_i \in \mathbb{R}$) and let $E = \{e_1, \ldots, e_l\}$ be the set of elements that we want to cover with the tests. Let matrix $M = \{m_{ik}\}$, of dimension $n \times l$, be defined as:

$$m_{ik} = \begin{cases} 1, & \text{if } e_k \text{ is covered by test } t_i. \\ 0, & \text{otherwise.} \end{cases}$$

The objective of minimization is to find a subset of tests $X \subseteq T$ with minimum cost such that all the elements in E are covered and the validated dependencies $D = \{d_{ij}\}$ between tests are respected. Formally:

$$\textit{minimize} \quad \sum_{i=1}^{n} c_i t_i \tag{6.3}$$

subject to:

$$\sum_{i=1}^{n} m_{ik} t_i \geq 1, \qquad 1 \leq k \leq l \tag{6.4}$$

$$\forall d_{ij} \in D,\ d_{ij} = 1 \wedge t_i = 1 \implies t_j = 1 \tag{6.5}$$

Equation 6.3 is the objective function. It is a linear combination of selected tests along with the related execution cost coefficients. The goal is to find a test suite having the smallest execution

M	e_1	e_2	e_3	e_4	e_5	e_6
t_1	1	0	1	1	0	0
t_2	0	1	1	1	0	0
t_3	1	0	0	0	1	0
t_4	1	1	0	0	1	0
t_5	0	0	1	1	1	0
t_6	0	0	0	0	0	1

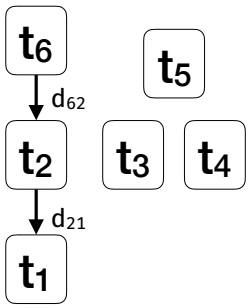


Figure 6.4: Minimization example

cost. Equation 6.4 represents the coverage constraints, one for each element e_k to be covered. Each coverage constraint specifies that at least one test covering each element e_k must be included in the final test suite X. Equation 6.5 represents the dependency constraints, one for each validated dependency $d_{ij} = 1$. The dependency constraint states that if a dependee test t_i is included in the final test suite X then its dependent t_j must be included as well.

For example, let us consider $T = \{t_1, t_2, t_3, t_4, t_5, t_6\}$ and $C = \{3.0, 3.0, 2.5, 3.5, 4.0, 1.0\}$. Figure 6.4 shows the coverage matrix M on the left hand side and the test dependency graph *TDG* on the right hand side. Below, we list both the coverage constraints, on the left hand side, and the dependency constraints, on the right hand side:

$$
\begin{aligned}
t_1 + t_3 + t_4 &\geq 1\ (e_1) \qquad & t_2 = 1 &\implies t_1 = 1\ (d_{21}) \\
t_2 + t_4 &\geq 1\ (e_2) \qquad & t_6 = 1 &\implies t_2 = 1\ (d_{62}) \\
t_1 + t_2 + t_5 &\geq 1\ (e_3, e_4) & & \\
t_3 + t_4 + t_5 &\geq 1\ (e_5) & & \\
t_6 &= 1\ (e_6) & &
\end{aligned}
$$

The first coverage constraint (top left) tells the solver that at least one of the tests $\{t_1, t_3, t_4\}$ must be included in the solution because those tests cover the element e_1. The first dependency constraint (top right) is related to the dependency d_{21} in the dependency graph shown in Figure 6.4. The dependency states that $t_2 \rightarrow t_1$, hence if t_2 is included in the solution ($t_2 = 1$) then the tests t_2 depends on, i.e. t_1, must be included as well ($t_1 = 1$).

The objective function for this problem instance is $(3.0t_1 + 3.0t_2 + 2.5t_3 + 3.5t_4 + 4.0t_5 + 1.0t_6)$. The only solution is given by $X = \{t_1, t_2, t_3, t_6\}$, where $t_i = 1,\ \forall t_i \in X$.

6.2.3 Implementation

We implemented our approach in a tool called DANTE (**D**ependency-**A**ware, Crawli**n**g-Based Web **T**est G**e**nerator) which is publicly available [1]. The tool is written in Java, and generates Selenium WebDriver Java web test suites. We used Crawljax 4.1 [MvDL12] for generating crawling-based test suites as input to our tool. DANTE integrates TEDD [2], to compute the test dependencies, the Java bindings of Z3 [z319] 4.8.4, a state-of-the-art SAT solver, to compute the minimization, and *NSGAII* [DPAM02] for the bi-objective optimization. The output of DANTE is a minimized test suite and a list of validated dependencies.

6.3 Empirical Evaluation

6.3.1 Research Questions

We conducted an empirical study to answer the following research questions:

RQ_1 (breakage rate). *What is the breakage rate of the segments generated by Crawljax?*

RQ_2 (minimization). *What is the test suite reduction rate achieved by* DANTE*?*

RQ_3 (performance). *What is the runtime of* DANTE *and of* DANTE*'s steps?*

RQ_4 (filtering heuristics). *What is the contribution of coverage-driven filtering and bi-objective filtering in making test dependency validation efficient?*

RQ_5 (coverage). *How does* DANTE *compare to web test generators based on a navigation model obtained through crawling in terms of client side code coverage and breakage rate?*

RQ_1 and RQ_2 are the core questions for the validation of the approach. In fact, DANTE produces minimized test suites with zero breakage rate (thanks to dependency analysis) by construction. So, it is important to understand the improvement achieved over the initial breakage rate, which might be non zero, and over the initial test suite size, which might be substantially larger than the final one. These two research questions assess the practical effects of dependency analysis and test suite minimization.

The next research question, RQ_3, is about the tool's performance. RQ_4 is about the contribution of the two proposed filters to the performance of DANTE. The final research question, RQ_5, compares the coverage achieved by our approach with that achieved by existing crawling and model based approaches (resp. Crawljax and Ext-Crawljax, our reimplementation of Atusa).

[1] `https://github.com/anon-icst2020/ICST20-submission-material-DANTE`
[2] `https://www.github.com/matteobiagiola/FSE19-submission-material-TEDD`

6.3.2 Subject Systems

For the evaluation of this work we took the subjects selected for our work on test case generation (see Section 4.8), except the subject *pagekit* [pag18]. For the sake of completeness, we report their features in the table below. The reason regarding our focus on SPA is given in Section 4.8.2 since also in the empirical evaluation in this chapter we measure code coverage.

Table 6.1: Experimental subjects

Subject	Framework	LOC (JS)	Stars	Commits	Last Commit
dimeshift [dim18]	Backbone	5,140	143	201	2019
splittypie [spl18]	Ember.js	2,710	75	331	2017
Phoenix Trello [pho18]	React	2,368	2,233	422	2016
retroboard [ret18]	React	2,144	420	487	2018
petclinic [Pet18]	AngularJS	2,939	80	78	2019

6.3.3 Procedure and Metrics

Procedure. First, we executed Crawljax on each subject system. We configured the crawler to run on the Chrome browser, with the default exploration strategy (depth-first visit) and state abstraction function (DOM string comparison), and a runtime limit of 30 minutes. Crawljax segments the sequence of crawled states following the strategy described in Section 6.1, and returns a test suite. Then, we manually fixed the flakiness of such test suite by adding delays where appropriate. We executed each test suite 10 times to check that identical outcomes are obtained across all executions.

To answer our research questions, we evaluated *two different configurations* of DANTE, by enabling in turn only one of the proposed filtering mechanisms, followed by test suite minimization (in both cases): (1) *coverage-driven filter & minimization*, and (2) *bi-objective filter & minimization*. We did not enable both filters at the same time because we experimentally found that the latter becomes useless if the former is applied.

Metrics. Concerning the *breakage rate* of the test segments generated by Crawljax ($\mathbf{RQ_1}$), we measured the number of generated test cases that *broke* when executed in isolation. To assess the *reduction rate* ($\mathbf{RQ_2}$), we measured the number of test cases in the minimized test suite produced by DANTE with respect to the original test suite generated by the crawler.

We evaluated *performance* ($\mathbf{RQ_3}$) by measuring the execution time (in minutes) of DANTE's components. To asses the impact of the *filtering heuristics* (i.e., coverage-driven filter vs bi-

objective filter) on the test dependency validation time ($\mathbf{RQ_4}$), we executed DANTE twice on each subject, by enabling in turn only one of the proposed filtering mechanisms, and compared the running time to complete the dependency validation in each configuration.

Concerning *coverage* ($\mathbf{RQ_5}$), we compared DANTE against Crawljax and Ext-Crawljax, our reimplementation of Atusa [MvD09], a state-of-the-art model-based web test generator, in terms of client-side coverage and breakage rate (only tests that do not break during execution are considered for coverage). We configured Ext-Crawljax to use the same inputs used during crawling following the indications of the authors of Atusa [MvD09]. To measure coverage, we used *cdp4j* 3.0.8 [cdp19], the Java implementation of Chrome DevTools [Chr19], which outputs the *byte coverage* of the executed client-side JavaScript code reached by each test case. The reason why we chose a different code coverage tool with respect to the empirical evaluation in the test case generation chapter in Section 4.8 is that the Chrome DevTools [Chr19] does not require any instrumentation of the JS code and it can be applied on any web application. On the other hand, the code coverage tool `Istanbul` [Ist18] cannot be easily applied to web application whose client side is written in TypeScript [3] (a typed superset version of JavaScript that compiles to plain JavaScript).

6.3.4 Results

$\mathbf{RQ_1}$ (brekage rate). Table 6.2 (macro-column Breakage Rate) shows the number of tests generated by Crawljax on each subject system within the 30 minutes time budget (Column 2). Columns 3-4 show the breakage rate of such test suites when test cases are executed in isolation, both numerically and percentage-wise. On average, 69 tests were generated by the crawler, of which 85% break when executed in isolation.

The minimum breakage rate occurred for *petclinic* (24 broken tests), whereas for the remaining subjects almost all test cases broke (96–98% breakage rates). These empirical results essentially show that tests generated from the output of a crawler cannot be executed without taking into account the hidden test dependencies among them, due to the shared web application states.

We manually investigated such high breakage rates and found that they are due to states created by the initial tests, which the next tests rely upon. For example, in *retroboard*, *Phoenix Trello* and *dimeshift*, the first tests perform a login operation; the next tests are supposed to be executed in a state (browser and database state) in which the login has already been performed. In *splittypie*, an expense splitting application, the second test creates an event for splitting expenses among participants. After the event is created, the event page view becomes the new application home page (a refresh redirects the browser to the event page), and all tests executed after the second one expect such page as starting page (unless a test case removes that event). *petclinic*'s tests have a lower breakage rate. Nevertheless, its tests still exhibit breakages due to shared states

[3] `https://www.typescriptlang.org/`

produced by previous tests (for instance, a certain pet must be created prior to schedule a visit with a veterinarian).

From our experiments, no test breakages occurred in test suites generated by DANTE, as test dependencies are revealed and each test is executed under the proper state preconditions. As such, in Table 6.2, we do not report the breakage rates of DANTE as they are all zero.

RQ_2 (minimization). Table 6.2 (macro-column Minimization) shows the minimization results achieved by both configurations of DANTE on our subjects. For each configuration, the table shows the number of tests removed at each step of the approach (coverage-based/bi-objective filter; test suite minimization), the final test suite size, and the minimization rate achieved with respect to the initial test suites.

In the first configuration (Coverage-driven filter & Minimization), the main contribution to minimization is given by the coverage-driven filter (columns 5–6), which removed 83% of the initial test cases on average, with minimization scores greater than 70% across our subjects. The greatest minimization score occurred for *retroboard* (95%), whereas the lowest occurred for *Phoenix Trello* (72%). SAT solver-based minimization (columns 7–8) reduced, on average, only by 1% the original test suite size. Therefore, in this configuration, the coverage-driven filter contributes substantially more than the minimization step to the reduction of the size of the initial test suites. Overall, in the first configuration, the final test suites generated by DANTE for our subjects are 84% smaller than the initial ones generated by Crawljax (columns 9–10). The biggest reduction occurred for *retroboard* (96%), whereas the lowest occurred for *Phoenix Trello* (72%).

In the second configuration (Bi-objective filter & Minimization), the SAT solver-based minimization is responsible for the *whole initial test suite reduction*, since the bi-objective filter does not eliminate tests but dependencies. The minimization step removed, on average, as much as 81% of the initial tests (columns 11–13). Similarly to the first configuration, the biggest reduction occurred for *retroboard* (96%), whereas the lowest occurred for *Phoenix Trello* (64%).

The two evaluated configurations of DANTE achieve similar minimization scores on our subject systems (84% *vs* 81%).

The difference is explained by the different instances of the problem that the SAT solver must solve, and by the functioning of TEDD, which does not ensure having a minimal test dependency graph. In particular, the set of dependency constraints is different among configurations because the two different filtering techniques produce two different initial test dependency graphs (in terms of number of tests, hence in terms of dependencies) as input for TEDD. Consequently, TEDD produces two slightly different, yet valid, test dependency graphs. Thus, the SAT solver formulation takes into account two different sets of dependency constraints.

Table 6.2: Results for Feasibility (RQ_1), Effectiveness (RQ_2), Performance (RQ_3) and Filtering Heuristics (RQ_4).

	BREAKAGE RATE			MINIMIZATION										PERFORMANCE											
				Coverage-driven Filter & Minimization						**Bi-objective & Minim.**				**Coverage-driven Filter & Minimization**				**Bi-objective Filter & Minimization**				**Relative Saving**			
	Generated Tests (#)	Broken Tests (#)	Breakage Rate (%)	Removed by Coverage-driven (#)	%	Removed by Minimization (#)	%	Final Num of Tests (#)	%	Removed by Minimization (#)	Final Num of Tests (#)	%		Coverage-driven Filter (min)	Dependency Validation (min)	Minimization (min)	Total (min)	Bi-objective Filter (min)	Dependency Validation (min)	Minimization (min)	Total (min)	Diff (min)	%	Speed-up (×)	
petclinic	65	24	37	54	83	0	0	11	83	56	9	86		50.2	25.6	0.3	76.1	23.5	1,005.6	0.3	1,005.9	953.4	93	13×	
splittypie	69	66	96	59	86	1	1	9	87	56	13	81		39.1	22.6	0.2	61.9	27.7	1,842.0	0.2	1,842.2	1,808.0	97	30×	
retroboard	100	98	98	95	95	1	1	4	96	96	4	96		0.5	1.7	0.6	2.8	97.2	2,026.8	0.6	2,027.4	2,096.4	99	730×	
Phoenix Trello	53	51	96	38	72	0	0	15	72	34	19	64		13.2	58.2	0.2	71.6	12.2	1,242.0	0.2	1,242.2	1,182.9	94	17×	
dimeshift	56	55	98	44	79	2	4	10	82	44	12	79		50.6	36.9	0.2	87.7	15.2	2,178.6	0.2	2,178.8	2,106.4	96	25×	
Average	69	59	**85**	58	83	1	1	10	**84**	57	11	**81**		30.7	29.0	0.3	**60.0**	35.2	1,659.0	0.3	**1,659.3**	1,629.4	95	**28X**	

RQ_3 (performance). Table 6.2 (macro column Performance) shows the running time (in minutes) of each step of DANTE for all subject systems and for both configurations.

The coverage-driven filter, which includes greedy coverage-driven test selection and test suite fixing (Algorithm 1), takes 30.7 min on average (columns 14–16). Most of such execution time is devoted to test suite fixing, whereas the cost of the greedy coverage-driven test selection is negligible (order of few seconds per subject). The fastest execution of the coverage-driven filter occurs for *retroboard* (25 seconds), while the slowest occurs for *dimeshift* (51 minutes).

On average, dependency validation takes as much as coverage-driven filtering (29 minutes), whereas the cost of the minimization step is negligible (18 seconds). Recall that the dependency validation runtime grows more than linearly with the number of dependencies in the test dependency graph, and such number is reduced due to filtering. For instance, in *Phoenix Trello*, 15 (53-38) tests are retained after the coverage-driven filter, whereas in *retroboard* only 5 are left (100-95). In the former case, dependency validation takes 58 minutes, whereas in the latter case it takes 2 minutes. Overall, the first configuration of DANTE takes on average 60 minutes to compute the final minimized test suite (column 17).

Columns 18–20 present the runtime results for each step of the second configuration, i.e., when the bi-objective filter is enabled. The bi-objective filter (Column 18) was configured with a population size of 100 and was granted 1 million fitness evaluations for each subject (such hyper-parameters have been fine tuned by means of a few preliminary runs of the algorithm). The runtime of the bi-objective filter depends on the number of dependencies (hence, on the initial number of tests). The slowest case occurs for *retroboard* (100 tests), in which the filter runtime takes 97 minutes (1.6 hours), whereas the fastest case occurs for *Phoenix Trello* (53 tests), with 12 min.

On average, the bi-objective filter runtime is 35 minutes across all subjects. The dependency validation runtime (Column 19) is as high as 27 hours, taking 16 hours in the best case (*petclinic*), and up to 36 hours in the worst case (*dimeshift*). Also in this configuration, the cost of the minimization step (Column 20) is negligible (18 seconds on average). Overall, the second configuration of DANTE takes on average 27 hours to compute the final minimized test suite, due to the high cost of the dependency validation on large test dependency graphs.

RQ_4 (filtering heuristics). Table 6.2 (macro column Relative Saving) compares the two configurations of DANTE further. Specifically, Columns 21-23 show the relative saving, in minutes and percentage-wise, of the first configuration with respect to the second configuration. Finally, Column 23 shows also the relative speed-up.

To fully highlight the importance of filtering the test dependencies prior to the minimization step, we recall that, when none of the two filters was active, the dependency validation never terminated within 48 hours (2 days) for all subject systems. So, it is critical to enable one of the two filters and among them, on average, the coverage-driven filter allows saving 26 hours with respect to the bi-objective filter, with a speed-up of 28×.

Table 6.3: Comparison between DANTE, Ext-Crawljax and Crawljax in terms of client-side byte coverage and breakage rate across all subjects (RQ_5).

	DANTE			Crawljax		Ext-Crawljax		
	Num of Tests (#)	Breakage Rate (%)	Coverage (%)	Num of Tests (#)	Coverage (%)	Num of Tests (#)	Breakage Rate (%)	Coverage (%)
petclinic	11	28.22	65	36.92	26.04	259	84.55	23.05
splittypie	9	24.32	69	95.65	15.18	54	9.26	21.41
retroboard	4	40.50	100	98.00	37.93	99	37.37	38.80
phoenix	15	53.93	53	96.22	36.21	57	42.10	47.33
dimeshift	10	42.07	56	98.21	26.53	73	91.32	29.37
Average	10	**37.81**	69	85.00	**28.38**	108	52.92	**31.99**

RQ_5 (coverage). Table 6.3 compares the best configuration of DANTE (Coverage-driven filter & Minimization) with Crawljax and Ext-Crawljax in terms of client-side byte coverage and breakage rate. On average, the final test suites generated by DANTE are composed of 10 tests, whereas those generated by Crawljax and Ext-Crawljax contain 69 and 128 tests, respectively (+590% and +1180%). Concerning the breakage rates, 53% of tests generated by Ext-Crawljax broke when executed in isolation, whereas *none* of the tests generated by DANTE breaks. This means that a substantial proportion of test cases generated by Ext-Crawljax has to be discarded because they are *infeasible*, hence those tests do not contribute to increase the coverage of the application under test. Overall, tests generated by DANTE have a coverage increase of 33% with respect to Crawljax, and 18% with respect to Ext-Crawljax.

6.3.5 Threats to Validity

Using a limited number of subject systems in our evaluation poses an *external validity* threat, in terms of generalizability of our results. We tried to mitigate this threat by choosing five single-page JavaScript web applications, developed with popular frameworks and pertaining to different domains, although more subject systems are needed to fully address the generalization threat.

Threats to *internal validity* might come from confounding factors of our experiments. We compared all competing algorithms under identical parameter settings. Our choice of Crawljax as baseline for crawling-based test suites might pose another threat, as well as Ext-Crawljax, which is also based on the navigational model provided by Crawljax. However, Crawljax is a notable, well-known and maintained research tool (it now comes with release 4.1), and, to our knowledge, no better alternatives have been proposed yet. For $\mathbf{RQ_5}$, we adopted a tool that computes *byte coverage*, instead of the classical statement or branch coverage. However, byte coverage is a fine-grained, precise coverage metric, which can be turned into more coarse-grained coverage metrics, e.g. line coverage, if needed. Moreover, the reimplementation of Atusa's extraction algorithm constitutes an internal validity threat. However, we followed the algorithm description reported in the Atusa's paper [MvD09] for our reimplementation.

With respect to *reproducibility*, we will make the source code of DANTE and all subject systems available [4], to ensure that the evaluation is repeatable and our results reproducible.

6.3.6 Discussion

Dependency and Redundancy. Our empirical results confirm that web tests generated from crawler's navigations often break because they involve hidden test dependencies. Moreover, from our experiments, most of such tests are redundant and can be removed without compromising coverage. DANTE was able to make all crawler-generated tests executable, reducing the breakage rate to zero. It does so by automatically detecting their dependencies and producing only test schedules that respect them. In our experiments, DANTE eliminated all redundant test cases in the initial test suites that do not contribute to coverage and can be safely removed since they do not involve any required dependency.

Crawler based and Model based Web Test Generation. Our empirical results show that our approach to web test generation outperforms both crawler based test generation, which is limited by the problems of test dependency and test redundancy, as well as model based web test generation, which is affected by path and input infeasibility. DANTE overcomes the limitations of both approaches by retaining the feasible inputs and sequences provided by a crawler, while fixing the test dependencies required to ensure feasibility and eliminating unnecessary test cases.

Filtering Techniques. Both evaluated configurations of DANTE have shown significant effectiveness ($\mathbf{RQ_2}$). Moreover, both proposed filtering techniques allowed reducing the cost of test dependency validation ($\mathbf{RQ_3}$), which is known to be in general a computationally expensive step, and our work makes no exception. To this aim, the coverage-driven filter has shown better results than the filter based on bi-objective optimization. The reason behind this is the huge size of the initial test dependency graph (on average, 2,453 dependencies). The coverage-driven fil-

[4] https://github.com/anon-icst2020/ICST20-submission-material-DANTE

ter allowed removing many false dependencies *prior* to the expensive validation phase, which explains the 28× time speed-up over the bi-objective filter.

Limitations. Our approach assumes that tests execute deterministically. DANTE does not include a procedure to automatically fix the flakiness of the test cases generated by the crawler, which is a non-trivial task. For instance, simply adding `wait` statements systematically within the test code may unnecessarily and artificially increase the runtime of the test suite, and it may not work when tests are executed on different browsers or hardware configurations. For such reasons, in our experiments, test flakiness was fixed manually after the crawling step.

Moreover, test cases generated by DANTE do not include any explicit functional oracle, such as test case assertions. Hence, only the implicit assertions (application crashes or runtime errors) can expose faults in the apps under test, unless the automatically generated tests are augmented with manually written assertions. It would be however possible to automatically generate assertions that capture the observed (instead of the intended) behaviour for regression testing.

Chapter 7

Conclusions and Future Work

7.1 Summary of Achievements

- **Model based test case generator.** We have proposed a diversity based approach for web test generation implemented in our tool DIG [1]. DIG can assess a high number of test case candidates without executing them in the browser, making test generation significantly more efficient than state-of-the-art techniques. Differently from search based approaches (i.e. SUBWEB), DIG is fully automated, and it does not require any specific guidance to generate feasible test cases. Our empirical evaluation on six real-world web applications shows that DIG achieves higher coverage and fault detection rates significantly earlier than crawling based (i.e. Atusa) and search based web test generators (i.e. SUBWEB). Regarding effectiveness there is no statistical evidence that DIG and SUBWEB are different in terms of coverage and fault detection; the difference is that DIG does not require the manual step of writing the guards which are needed for SUBWEB.

- **Approach to detect dependencies in E2E web test suites.** We introduced a novel notion of persistent read-after-write (PRAW) dependencies, which extends the standard notion of RAW dependencies to the web domain. Then, we proposed a test dependency technique for E2E web test cases based on string analysis and NLP implemented in a tool called TEDD [2]. TEDD achieves an optimal trade off between false dependencies to be removed and missing dependencies to be recovered in six web test suites. Specifically, our results show that TEDD can correctly detect and validate test dependencies up to 72% faster than the baseline in which the graph contains all possible dependencies associated with the original test ordering. The test dependency graphs produced by TEDD enable test parallelization, with a speed-up factor of up to 7×.

[1] https://github.com/matteobiagiola/FSE19-submission-material-DIG
[2] https://github.com/matteobiagiola/FSE19-submission-material-TEDD

- **Turn web crawler into a test generator.** Web crawlers have long been adopted to generate test cases for web applications, mostly following a model based approach. We show that the raw output of a crawler, i.e., a navigation sequence, cannot be easily turned into individual test cases that can be replayed as-is, since 85% of them fail when executed in isolation. To this aim, we propose a novel approach to web test generation, implemented in a tool called DANTE [3], that transforms the output of a crawler into executable test cases. DANTE analyzes the test sequences produced by a crawler and determines the test dependencies occurring between pairs of test cases. It also removes the redundant tests by means of a coverage based pre-filter and a SAT based minimization step. Our experimental results show that test suites generated by DANTE are 84% smaller on average than the ones produced by the crawler, and never exhibit test failures. Our results also show that DANTE outperforms a state-of-the-art model based test generator (i.e. Atusa) in terms of coverage and failure rate.

7.2 Discussion

With respect to the problem statement (see Section 1.2) we explored three ways to address it. Table 7.1 shows a qualitative analysis that compare the three approaches along different dimensions.

The first dimension we consider is how the different approaches handle the feasibility problem (first column in Table 7.1). SUBWEB uses a search based approach to guide the search towards feasible individuals, whereas DIG uses diversity to explore the space of feasible test paths at large and execute only those that are *far* from the already executed ones. Moreover, our results show that promoting diversity is beneficial not only to a thorough exploration of the application behaviours, but also to the feasibility of automatically generated test cases. On the other hand, DANTE uses a crawling based approach to generate test cases bypassing the construction of the model of the web application under test. While the model based approaches of SUBWEB and DIG have to cope with the feasibility problem, DANTE is not affected by it, since it directly generates concrete test cases.

Regarding the test quality assessment point of view (second column of Table 7.1), both SUBWEB and DIG have quality metrics that guide the search/exploration towards feasible paths. In particular SUBWEB uses the guards or preconditions of each PO method (e.g. transition of the navigation graph) to understand how good is a test case with respect to another. In fact, guards define the fitness function that guides the search towards feasible paths. A test is as good as it is close to satisfy the conditions specified in each PO method. Differently from SUBWEB, DIG establishes the quality of a test case by computing its distance with respect to already executed test cases. The best test case is the farthest from those already executed. On the other hand,

[3] `https://github.com/anon-icst2020/ICST20-submission-material-DANTE`

Table 7.1: Qualitative comparison between SUBWEB, DIG and DANTE.

Approach	Feasibility	Test Quality Assessment	Automation	Efficiency	Independent Tests
SUBWEB	Yes (search)	Yes (guards/feasibility)	No (guards)	No (execution)	Yes
DIG	Indirectly (diversity)	Yes (distance metric)	Yes	Yes (static comparison)	Yes
DANTE	Not affected (crawling)	No (no quality metric)	Yes	Yes	No

DANTE does not use any quality metric to decide whether a test case is better than another. In fact, DANTE uses a web crawler whose exploration of the web application under test is guided by the state abstraction function and by the unfired candidate actions in each state.

Considering the automation point of view (third column of Table 7.1), SUBWEB relies on preconditions of PO methods which depend on the business logic of the web application under test, hence they cannot be generated fully automatically. DIG does not need preconditions to evaluate the distance between test cases, as well as DANTE where every step is automatic (except for the configuration of the crawler which may need to be done manually for each web application in order to achieve a thorough exploration). In the automation point of view we do not consider the flakiness problem which affects E2E test cases. In fact, the three approaches presented in this thesis do not tackle the flakiness problem, which needs to be solved manually. In DIG and SUBWEB, which are PO based test generators, the process of fixing flaky tests is easier for developers since flakiness can be fixed at the PO level. On the other hand, DANTE generates test cases composed of bare Selenium statements, therefore if a test is flaky it may need to be fixed multiple times.

Another interesting dimension to explore is efficiency (fourth column of Table 7.1), which we consider as the number of executions needed for feasibility assessment. The reason is that, in the context of E2E web testing, the execution of a test case is computationally expensive since the test execution environment is the browser. SUBWEB directly evaluates feasibility and, in order to establish the feasibility of a test, it needs to execute it. Every generated test is executed and, if the test is infeasible, the execution time is wasted because the test does not contribute to increase the coverage of the navigation graph. Therefore, the search based approach employed by SUBWEB is not efficient. DIG, instead, evaluates diversity which can be seen as a surrogate of feasibility and, most importantly, it can be computed statically. DIG generates more test cases than SUBWEB but it only executes the most diverse with respect to those already executed, therefore it is an efficient method. DANTE does not need to assess feasibility since each generated test is executed but it is feasible by construction (i.e. it is concrete). Therefore, if efficiency is a measure of feasibility assessment, DANTE is an efficient method.

Both DIG and SUBWEB generate independent test cases (fifth column of Table 7.1). Each test exercises its own scenario and it does not depend on the execution of other tests, since the state

of the web application under test is reset to its starting condition after each test execution. On the contrary, DANTE generates dependent tests because it is based on a web crawler. After crawling DANTE detects the dependencies and minimizes the generated test suite. The drawback of dependency aware test generators is that test optimization techniques, such as test parallelization, test minimization, test prioritization and test selection, have been formulated for independent test suites and they need to be reformulated to take into account the detected dependencies.

To conclude, DIG and SUBWEB are model based approaches and the model represents the high level functionalities of the web application under test, while DANTE resorts to a web crawler, which generates tests by randomly exploring the web application. The other important difference is that DANTE does not require manual intervention (except for crawling configuration) whereas DIG and SUBWEB can either resort to a web crawler to extract the navigation graph of the web application or to manually written POs from which the navigation graph can be extracted. If manually written POs are available DIG and SUBWEB are preferable to DANTE, since the navigation graph extracted from manually written POs contains all the functionalities that are worth testing, whereas the exploration performed by DANTE can be incomplete (i.e. it is limited by the exploration strategy of the web crawler and by its state abstraction function). In such context, DIG is preferable to SUBWEB since PO methods preconditions are usually not written by PO developers and it is difficult to extract them automatically. On the other hand, if manually written POs are not available both DIG and DANTE are based on the performance of a web crawler. Particularly, DIG is less sensitive to the web crawler, since it is sufficient that the crawler explores a certain state for the respective PO to be created by Apogen. In fact Apogen, that generates the PO classes from the output of the web crawler, creates the PO methods (e.g. form submission, transition to other POs) independently from the fact that those actions have been performed during crawling. On the contrary, tests generated by DANTE will not exercise specific parts of the web application if the corresponding actions have not been fired during crawling. However, since DIG is based on Apogen, it is affected by its limitations, for instance clustering limitations that could result in state/PO redundancy and not meaningful DOM elements ids which create not meaningful method names. Those issues may need to be addressed manually, whereas DANTE can be seen as a completely automatic approach.

7.3 Future Work

Empirical comparison between DIG and DANTE. In order to evaluate which approach performs better, we plan to carry out an empirical comparison between the model based approach used by DIG with the crawling based approach DANTE resorts to. A way to compare them is to measure effectiveness and efficiency in terms of client side code coverage, since DANTE does not use a navigation graph. In order for the comparison to be fair the crawling timeout for both approaches should be the same. Furthermore, the time taken by DIG to cover all the transitions of the PO based navigation graph should be measured, together with the time taken

by DANTE to perform the filtering, validation and minimization steps. Finally, the time to fix the POs generated by Apogen, if any, should also be measured.

Crawling. An important crawling parameter that can significantly impact the exploration of the crawler (in particular Crawljax [MvDL12]) given a target web application, is the exploration strategy, i.e. the strategy that decides which state to expand next at each crawling step. The feedback-directed exploration proposed by Fard et al. [MFM13] incorporates different coverage metrics to guide the exploration. Experimental results are promising w.r.t extensive exploration strategies such as depth-first, breadth-first and random. Another interesting direction to explore is the use of search based techniques to guide the crawler during the exploration and compare them with the feedback-directed technique. A multi-objective algorithm has been applied successfully to automatically explore the behaviours of Android applications [MHJ16]. Promising results can be envisioned also in the context of web applications. Improving the exploration strategy would exercise more thoroughly the web application under test with the possibility of covering more functionalities and, hence, finding more bugs.

Automatic Test Suite Augmentation. An interesting line for further research is that of automatic test suite augmentation, i.e. an approach to test generation that considers code changes relative to the web application under test and their effects on past test suites. Test suite augmentation often implies fixing test cases that break in previously available test suites and producing new test cases that exercise new behaviours. Moreover, it can exploit information available from existing test suites for a previous version of the web application under test (i.e., *seeding*). Two directions can be promising, one more manual and straightforward, the other more automated and more challenging. The manual direction is a direct extension of SUBWEB and DIG. Indeed, in both cases the test generation process starts from a model of the web application under test built with POs. When the web application under test changes, the POs need to be manually modified by testers to address those changes. The test suite generated for the previous version is seeded into the test generation algorithm (either search based or diversity based) and executed. The tests that break would be discarded while the others constitute a good starting point to start exploring new behaviours.

The automated direction regards the use of a web crawler, in particular of DANTE. The idea would be to re-execute the test suite generated for the previous version (generated in a dependency aware manner), in order to build a model of the web application under test after the changes [MFMM14]. Then, the new model can be expanded using any exploration strategy (exhaustive, feedback-directed, search based), in order to exercise new behaviours. Finding effective techniques for test suite augmentation could improve the efficiency and the effectiveness of test case generation, especially in the context of continuous integration [CAFA14].

Test Dependency Detection. In the research line of test dependency detection we are currently studying the cost-effectiveness of TEDD during test suite evolution. The hypothesis is that dependency detection, as proposed in TEDD, is expensive ($\approx$ hours) only when no test dependencies are known. Given the validated dependency graph of a test suite of a version x of the web

application under test, the objective is to study the runtime of TEDD when the web application evolves (version $y > x$), which implies changes in the test suite. In particular, test cases that break have to be fixed or removed (if a certain functionality is removed), and new test cases have to be written for functionalities added in the new version. The incremental version of TEDD should, given the validated dependency graph for version x and the evolved test suite for version y, generate the validated dependency graph for version y.

Besides evaluating the cost-effectiveness of TEDD upon test suite evolution we can improve the dependency detection process to make it more efficient. For example we can define new methods to find dependencies between tests before the actual validation (the expensive process) starts. One possibility is to analyze the requests the REST client is making towards the REST server during test case execution. By the analysis of the HTTP request verb, the body of the request and the server response, it is possible to extract some information related to what the test is doing (reading/writing) and its relation with other tests. The advantage is that this heuristic can be applied to automatically generated tests but, on the other hand, it will be effective only with those web applications that implement the REST protocol.

Another direction regards the actual validation of the dependency graph, regardless of how those dependencies are extracted. The validation algorithm implemented in TEDD validates one dependency at a time. In order to improve efficiency, the validation can be parallelized by analyzing dependencies that belong to different weakly connected components in parallel. The parallelization algorithm should also take into account that dependencies can be added automatically (and not only removed) during the validation. The initial dependency graph may indeed have false negatives (missing true dependencies), which are added back by TEDD's recovery procedure. Speeding up the test dependency detection would also bring benefits to the dependency aware test case generation in which dependency detection is a necessary precondition.

Bibliography

[AB11] Andrea Arcuri and Lionel Briand. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 265–275. ACM, 2011.

[AB14] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[ACA12] Franco Arito, Francisco Chicano, and Enrique Alba. On the application of sat solvers to the test suite minimization problem. In *International Symposium on Search Based Software Engineering*, pages 45–59. Springer, 2012.

[ADJ+11] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 571–580, New York, NY, USA, 2011. ACM.

[AH11] Nadia Alshahwan and Mark Harman. Automated web application testing using search based software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 3–12. IEEE Computer Society, 2011.

[AIB10] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Formal analysis of the effectiveness and predictability of random testing. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 219–230. ACM, 2010.

[aja05] Ajax: A new approach to web applications. adaptive path. `https://immagic.com/eLibrary/ARCHIVES/GENERAL//ADTVPATH/A050218G.pdf`, 2005.

[AKD+08] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In *Proceedings*

of the 2008 international symposium on Software testing and analysis, ISSTA '08, pages 261–272, New York, NY, USA, 2008. ACM.

[Arc13] Andrea Arcuri. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability*, 23(2):119–147, 2013.

[Arc17] Andrea Arcuri. Many independent objective (MIO) algorithm for test suite generation. In *Search Based Software Engineering - 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings*, pages 3–17, 2017.

[Ava14] Satya Avasarala. *Selenium WebDriver practical guide*. Packt Publishing Ltd, 2014.

[Bin96] Robert V. Binder. Testing object-oriented software: a survey. *Software Testing, Verification and Reliability*, 6(3-4):125–252, 1996.

[BK14] Jonathan Bell and Gail Kaiser. Unit test virtualization with vmvm. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 550–561. ACM, 2014.

[BKMD15] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. Efficient dependency detection for safe java test acceleration. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 770–781. ACM, 2015.

[BWK05] Stefan Berner, Roland Weber, and Rudolf K. Keller. Observations and lessons learned from automated testing. In *Proceedings of 27th International Conference on Software Engineering*, ICSE 2005, pages 571–579. IEEE Computer Society, 2005.

[CAFA14] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. Continuous test generation: enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 55–66. ACM, 2014.

[CCC+13] TY Chen, I Clark, MB Cohen, W Grieskamp, et al. An orchestrated survey on automated software test case generation. *Journal of Systems and Software*, 2013.

[CCT06] Kwok Ping Chan, TY Chen, and Dave Towey. Forgetting test cases. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 485–494. IEEE, 2006.

[cdp19] Chrome devtools protocol for java. `https://github.com/webfolderio/cdp4j`, 2019.

[CGK+11] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1066–1071. IEEE, 2011.

[Chr19] Css and js code coverage. `https://developers.google.com/web/updates/2017/04/devtools-release-notes#coverage`, 2019.

[CKL09] Tsong Yueh Chen, Fei-Ching Kuo, and Huai Liu. Adaptive random testing based on distribution metrics. *Journal of Systems and Software*, 82(9):1419–1433, 2009.

[CKM06] Tsong Yueh Chen, Fei-Ching Kuo, and Robert Merkel. On the statistical properties of testing effectiveness measures. *Journal of Systems and Software*, 79(5):591–601, 2006.

[CKMN04] Tsong Yueh Chen, F-C Kuo, Robert G Merkel, and Sebastian P Ng. Mirror adaptive random testing. *Information and Software Technology*, 46(15):1001–1010, 2004.

[CKMT10] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[CLM04] Tsong Yueh Chen, Hing Leung, and IK Mak. Adaptive random testing. In *Annual Asian Computing Science Conference*, pages 320–329. Springer, 2004.

[CLOM08] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: adaptive random testing for object-oriented software. In *Proceedings of the 30th international conference on Software engineering*, pages 71–80. ACM, 2008.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (2nd ed.)*. MIT Press and McGraw-Hill, 2001.

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

[Cor19] Stanford corenlp – natural language software. `https://stanfordnlp.github.io/CoreNLP/`, 2019.

[CRU19] Create, read, update and delete. `https://en.wikipedia.org/wiki/Create,_read,_update_and_delete`, 2019.

[DDM06] Bruno Dutertre and Leonardo De Moura. A fast linear-arithmetic solver for dpll (t). In *International Conference on Computer Aided Verification*, pages 81–94. Springer, 2006.

[DHHG06] Karnig Derderian, Robert M Hierons, Mark Harman, and Qiang Guo. Automated unique input output sequence generation for conformance testing of fsms. *The Computer Journal*, 49(3):331–344, 2006.

[Dij59] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[dim18] DimeShift: easiest way to track your expenses. `https://github.com/jeka-kiselyov/dimeshift`, 2018.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[DNSVT07] Arilo C Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, pages 31–36. ACM, 2007.

[DPAM02] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[ERKI05] Sebastian Elbaum, Gregg Rothermel, Srikanth Karre, and Marc Fisher II. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, 2005.

[ES+03] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[FA13] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276 –291, feb. 2013.

[FG99] Mark Fewster and Dorothy Graham. *Software Test Automation: Effective Use of Test Execution Tools*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[FPCY16] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. Test set diameter: Quantifying the diversity of sets of test cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 223–233. IEEE, 2016.

[FT00] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Doctoral dissertation, 2000.

[GBZ18] Alessio Gambi, Jonathan Bell, and Andreas Zeller. Practical test dependency detection. In *Proceedings of the 2018 IEEE Conference on Software Testing, Validation and Verification*, pages 1–11, April 2018.

[GD07] Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and arrays. In *International Conference on Computer Aided Verification*, pages 519–531. Springer, 2007.

[GEM15] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 211–222, New York, NY, USA, 2015. ACM.

[Git12] Test execution order in junit 4.11. `https://github.com/junit-team/junit4/blob/master/doc/ReleaseNotes4.11.html`, 2012.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[Goo14] Testing on the toilet: Writing descriptive test names. `https://testing.googleblog.com/2014/10/testing-on-toilet-writing-descriptive.html`, 2014.

[Goo19] Google java style guide. naming convention for junit tests. `https://google.github.io/styleguide/javaguide.html#s5.2.3-method-names`, 2019.

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

[Har11] Mark Harman. Software engineering meets evolutionary computation. *Computer*, 44(10):31–39, 2011.

[HJ01] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001.

[HJL+01] Mary Jean Harrold, James A Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. In *ACM Sigplan Notices*, volume 36, pages 312–326. ACM, 2001.

[HJZ15] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12. IEEE, 2015.

[HKL+10] Mark Harman, Sung Gon Kim, Kiran Lakhotia, Phil McMinn, and Shin Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 182–191. IEEE, 2010.

[HM09] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2009.

[HN15] Kim Herzig and Nachiappan Nagappan. Empirically detecting false test alarms using association rules. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 39–48, Piscataway, NJ, USA, 2015. IEEE Press.

[HRS16] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. WATERFALL: An incremental approach for repairing record-replay tests of web applications. In *Proceedings of 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '16, pages 751–762. ACM, 2016.

[Inv05] Inversion of control. `https://martinfowler.com/bliki/InversionOfControl.html`, 2005.

[Ist18] Istanbul: JavaScript test coverage made simple. `https://istanbul.js.org`, 2018. Accessed: 2018-08-01.

[JS-18] Front-end JavaScript frameworks. `https://github.com/collections/front-end-javascript-frameworks`, 2018.

[jso19] Json vs xml. `https://www.w3schools.com/js/js_json_xml.asp`, 2019.

[JUn19] Junit5 test execution order. `https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-execution-order`, 2019.

[Kap16] Sebastian Kappler. Finding and breaking test dependencies to speed up test execution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1136–1138. ACM, 2016.

[Kin75] James C King. A new approach to program testing. In *ACM SIGPLAN Notices*, volume 10, pages 228–233. ACM, 1975.

[Kor90] Bogdan Korel. Automated software test data generation. *IEEE Transactions on software engineering*, 16(8):870–879, 1990.

[Kor04] O. Koresteleva. *Nonparametric Methods in Statistics with SAS Applications*. CRC Press, Boca Raton, FL, 2004.

[LAG14] Guodong Li, Esben Andreasen, and Indradeep Ghosh. Symjs: automatic symbolic testing of javascript web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 449–459. ACM, 2014.

[LCRT13] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Proceedings of 20th Working Conference on Reverse Engineering*, WCRE '13, pages 272–281. IEEE Computer Society, 2013.

[LCRT16] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Approaches and tools for automated end-to-end web testing. *Advances in Computers*, 101:193–237, 2016.

[LDD14] Yuan-Fang Li, Paramjit K Das, and David L Dowe. Two decades of web application testing—a survey of recent advances. *Information Systems*, 43:20–54, 2014.

[Lee61] Chin Yang Lee. An algorithm for path connections and its applications. *IRE transactions on electronic computers*, (3):346–365, 1961.

[Lev66] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

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

[lib10] Why libraries are better than frameworks. `http://tom.lokhorst.eu/2010/09/why-libraries-are-better-than-frameworks`, 2010.

[LSRT15] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Using multi-locators to increase the robustness of web test cases. In *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation*, ICST '15, pages 1–10. IEEE, 2015.

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

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

[LTCZ09] Yu Lin, Xucheng Tang, Yuting Chen, and Jianjun Zhao. A divergence-oriented approach to adaptive random testing of java programs. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 221–232. IEEE, 2009.

[Luk13] Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[LZE15] Wing Lam, Sai Zhang, and Michael D. Ernst. When tests collide: Evaluating and coping with the impact of test dependence. Technical Report UW-CSE-15-03-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, March 2015.

[M+12] Tim Miller et al. Using dependency structures for prioritization of functional test suites. *IEEE transactions on software engineering*, 39(2):258–275, 2012.

[MC13] Atif M Memon and Myra B Cohen. Automated testing of gui applications: models, tools, and controlling flakiness. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1479–1480. IEEE Press, 2013.

[McM04] Phil McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[Mes15] Ali Mesbah. *Advances in Testing JavaScript-based Web Applications*, volume 97 of *Advances in Computers*, chapter 5, pages 201–235. Elsevier, 2015.

[MFM13] Amin Milani Fard and Ali Mesbah. Feedback-directed exploration of web applications to derive test models. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 278–287. IEEE Computer Society, 2013.

[MFMM14] Amin Milani Fard, Mehdi Mirzaaghaei, and Ali Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 67–78. ACM, 2014.

[MGN+17] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming google-scale continuous testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP '17, pages 233–242, Piscataway, NJ, USA, 2017. IEEE Press.

[MHJ16] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105. ACM, 2016.

[Mil95] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

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

[MT11] Alessandro Marchetto and Paolo Tonella. Using search-based algorithms for ajax event sequence generation during testing. *Empirical Software Engineering*, 16(1):103–140, 2011.

[MTR08] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. State-based testing of ajax web applications. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 121–130, Washington, DC, USA, 2008. IEEE Computer Society.

[MTR12] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. Reajax: a reverse engineering tool for ajax web applications. *IET software*, 6(1):33–49, 2012.

[MvD09] Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[MvDL12] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1):3:1–3:30, 2012.

[MvDR12] Ali Mesbah, Arie van Deursen, and Danny Roest. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering*, 38(1):35–53, 2012.

[OJPM17] Frolin S Ocariza Jr, Karthik Pattabiraman, and Ali Mesbah. Detecting unknown inconsistencies in web applications. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 566–577. IEEE Press, 2017.

[pag18] Pagekit: modular and lightweight CMS. `https://github.com/pagekit/pagekit`, 2018.

[Pet18] Angular version of the Spring PetClinic web application. `https://github.com/spring-petclinic/spring-petclinic-angular`, 2018.

[pho18] Phoenix: Trello tribute done in Elixir, Phoenix Framework, React and Redux. `https://github.com/bigardone/phoenix-trello`, 2018.

[PKT15] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–10, 2015.

[PKT18a] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2018.

[PKT18b] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Incremental control dependency frontier exploration for many-criteria test case generation. In *International Symposium on Search Based Software Engineering*, pages 309–324. Springer, 2018.

[PLEB07] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*, pages 75–84. IEEE Computer Society, 2007.

[PMP+15] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, 46:1155–1179, 2015.

[PPM04] Ted Pedersen, Siddharth Patwardhan, and Jason Michelizzi. Wordnet:: Similarity: measuring the relatedness of concepts. In *Demonstration papers at HLT-NAACL 2004*, pages 38–41. Association for Computational Linguistics, 2004.

[PY08] Mauro Pezze and Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[PZ17] F. Palomba and A. Zaidman. Does refactoring of test smells induce fixing flaky tests? In *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution*, ICSME 2017, pages 1–12, Sep. 2017.

[RCV+15] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering*, pages 93–108. Springer, 2015.

[ret18] Retrospective Board. `https://github.com/antoinejaussoin/retro-board`, 2018.

[RHVRH02] Gregg Rothermel, Mary Jean Harrold, Jeffery Von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[Ric04] Filippo Ricca. Analysis, testing and re-structuring of web applications. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 474–478. IEEE, 2004.

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

[RUCH01] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948, 2001.

[SJR+15] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 201–211. IEEE, 2015.

[SKBG13] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498. ACM, 2013.

[SLRT17] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. APOGEN: Automatic Page Object Generator for Web Testing. *Software Quality Journal*, 25(3):1007–1039, September 2017.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 263–272. ACM, 2005.

[spl18] Splittypie: easy expense splitting. `https://github.com/cowbell/splittypie`, 2018.

[STM12] Ali Shahbazi, Andrew F Tappenden, and James Miller. Centroidal voronoi tessellations-a new approach to random testing. *IEEE Transactions on Software Engineering*, 39(2):163–183, 2012.

[SWH11] Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl. Programs, tests, and oracles: The foundations of testing revisited. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 391–400, New York, NY, USA, 2011. ACM.

[SYM18] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. Visual web test repair. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '18. ACM, 2018.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[ted19] Web test dependency detection using nlp. `https://github.com/matteobiagiola/FSE19-submission-material-TEDD`, 2019.

[Tes19] Testng documentation. `https://testng.org/doc/documentation-main.html#annotations`, 2019.

[TLS+13] Suresh Thummalapenta, K. Vasanta Lakshmi, Saurabh Sinha, Nishant Sinha, and Satish Chandra. Guided test generation for web applications. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 162–171, Piscataway, NJ, USA, 2013. IEEE Press.

[TMN+12] Paolo Tonella, Alessandro Marchetto, Cu Duy Nguyen, Yue Jia, Kiran Lakhotia, and Mark Harman. Finding the optimal balance between over and under approximation of models inferred from execution logs. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference On*, pages 21–30. IEEE, 2012.

[TNM+13] Paolo Tonella, Cu Duy Nguyen, Alessandro Marchetto, Kiran Lakhotia, and Mark Harman. Automated generation of state abstraction functions using data invariant inference. In *Automation of Software Test (AST), 2013 8th International Workshop on*, pages 75–81. IEEE, 2013.

[Ton04] Paolo Tonella. Evolutionary testing of classes. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 119–128. ACM, 2004.

[TRM14] Paolo Tonella, Filippo Ricca, and Alessandro Marchetto. Recent advances in web testing. In *Advances in Computers*, volume 93, pages 1–51. Elsevier, 2014.

[TTN14] Paolo Tonella, Roberto Tiella, and Cu Duy Nguyen. Interpolated n-grams for model based testing. In *Proceedings of the 36th International Conference on Software Engineering*, pages 562–572. ACM, 2014.

[UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[vD15] Arie van Deursen. Beyond page objects: Testing web applications with state objects. *ACM Queue*, 13(6):20, 2015.

[vDMBK02] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring test code. In *Extreme Programming Perspectives*, pages 141–152. Addison-Wesley, 2002.

[VPK04] Willem Visser, Corina S Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 97–107. ACM, 2004.

[VSM18] Arash Vahabzadeh, Andrea Stocco, and Ali Mesbah. Fine-grained test minimization. In *Proceedings of the 40th International Conference on Software Engineering*, pages 210–221. ACM, 2018.

[WB04] Joachim Wegener and Oliver Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Genetic and Evolutionary Computation Conference*, pages 1400–1412. Springer, 2004.

[WC80] Lee J White and Edward I Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, (3):247–257, 1980.

[WG98] Joachim Wegener and Matthias Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15(3):275–298, 1998.

[WP94] Zhibiao Wu and Martha Palmer. Verbs semantics and lexical selection. In *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, pages 133–138. Association for Computational Linguistics, 1994.

[YH12] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[YM07] Xun Yuan and Atif M. Memon. Using GUI run-time state as feedback to generate test cases. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 396–405, Washington, DC, USA, May 23–25, 2007. IEEE Computer Society.

[YMZ15] Bing Yu, Lei Ma, and Cheng Zhang. Incremental web application testing using page object. In *Proceedings of the 2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, HOTWEB '15, pages 1–6, Washington, DC, USA, 2015. IEEE Computer Society.

[z319] The z3 theorem prover. `https://github.com/Z3Prover/z3`, 2019.

[Zel17] Andreas Zeller. Search-based testing and system testing: A marriage in heaven. In *Proceedings of 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing*, SBST '17, pages 49–50, May 2017.

[ZJW+14] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanç Muslu, Wing Lam, Michael D. Ernst, and David Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 385–396, New York, NY, USA, 2014. ACM.