CHAPTER THREE

Three Open Problems in the Context of E2E Web Testing and a Vision: NEONATE

Filippo Ricca*, Maurizio Leotta*, Andrea Stocco†
*DIBRIS, Università di Genova, Genova, Italy
†University of British Columbia, Vancouver, BC, Canada

Contents

Advances in Computers, Volume 113
ISSN 0065-2458
https://doi.org/10.1016/bs.adcom.2018.10.005

Abstract

Web applications are critical assets of our society and thus assuring their quality is of undeniable importance. Despite the advances in software testing, the ever-increasing technological complexity of these applications makes it difficult to prevent errors.

In this work, we provide a thorough description of the three open problems hindering web test automation: fragility problem, strong coupling and low cohesion problem, and incompleteness problem. We conjecture that a major breakthrough in test automation is needed, because the problems are closely correlated, and hence need to be attacked together rather than separately. To this aim, we describe NEONATE, a novel integrated testing environment specifically designed to empower the web tester.

Our utmost purpose is to make the research community aware of the existence of the three problems and their correlation, so that more research effort can be directed in providing solutions and tools to advance the state of the art of web test automation.

1. INTRODUCTION

Web applications pervade the lives of billions of individuals and have a significant impact on all aspects of our society, being crucial for a multitude of economic, social, and educational activities. Indeed, a considerable slice of modern software consists of web applications executed in the browser, running both on desktop and on smartphone devices [1]. Associations, enterprises, governmental organizations, companies, scientific groups use the web as a powerful and convenient way to promote activities/products and to carry out their core business. People daily use online services as source of information, means of communication, source of entertainment, and venue for commerce [2].

Hence, the quality and correctness of web applications are of undeniable importance. However, developing and maintaining a complex web system has become challenging, because the advances of web technologies in the last decade have also changed the way in which software running on the web is developed and maintained [3]. Unfortunately, the evolution of the tools for the analysis and testing of such complex software systems is not proceeding at the same pace.

Functional testing is one of the main approaches for assuring the quality of web applications. The goal of functional testing is to exercise the web

application under test to detect failures, where a failure can be considered as a deviation from the system's intended behavior. In many software projects, functional testing is neglected because of time or cost constraints [4]. However, the impact of failures in a web application may range from simple inconveniences (e.g., a malfunction that causes users dissatisfaction), up to huge economic problems (e.g., interruption of business).

Today, web developers mostly test their applications manually, i.e., they manually interact with the web applications to check if they behave as expected. Unfortunately, this practice is prone to errors, time-consuming, and ultimately not very effective. For these reasons, most teams try to automate manual testing activities by means of test automation tools. This process involves a manual step which consists of implementing the *test code* able to instrument the web application and run predefined test-oriented user scenarios. Test code provides input data, operates on GUI components, and retrieves information to be compared with oracles (e.g., using assertions).

In the web domain, test automation tools usually operate at user interface (GUI) level, interacting with the web elements that are displayed on the web page, as seen by the end users. This kind of testing is called end-to-end (E2E), because the application is tested as a whole, in its entirety, and from the perspective of the end user. Different categories of E2E test automation tools are available on the marketplace. For example, DOM-based tools use objects in the Document Object Model (DOM), the hierarchical structure underlying a HTML page, to locate and interact with web elements. Such tools have reached a high level of maturity and popularity as, for example, the case of Selenium [5]. A competing category of visual tools has also appeared in the last years based on the use of image recognition techniques to identify the web elements (e.g., JAutomate [6] and Sikuli [7]).

The economic convenience of test automation is strongly correlated to the maintenance cost of the test suites [8]. We have identified two major problems that contribute to increase such a cost. First, all the existing test automation technologies suffer—in different proportions—from the *fragility problem*. When a web application evolves to accommodate requirements changes or functionality extensions, existing automated test code can easily break, and testers must correct it. This task is expensive, because it is usually performed manually. Often breakages occur in response to minor changes, e.g., a change of the page layout only. In these cases the test code is named fragile.

Second, test code is usually strongly coupled with the application under test, and full of technical details that limit greatly its readability and an easy maintenance. This is the *strong coupling and low cohesion problem*.

Moreover, automated test suites are inherently incomplete, because they cover only a subset of the input and functionalities space of the application. This third limitation, which we called *incompleteness problem*, is an open testing problem. As we shall see in this work, the presence of these three problems makes the activity of testing adequately the web applications challenging.

Existing test automation tools offer little to no help to overcome the three aforementioned problems. A huge amount of resources is still required by the testers to cope with the creation and maintenance of a web test suite. Thus, we believe a paradigm shift is necessary to advance in the state of the art of web test automation. To this aim, we envision NEONATE (Novel algorithms/techniques for building and maintaining Web Test Code), an integrated testing environment able to empower the web tester limiting the three open problems. We have envisioned NEONATE out of the knowledge gained during industrial collaborations and the desire to solve concrete problems [9, 10].

The contributions of this work are as follows:

- a detailed description of three big open problems in the context of web test automation, namely, fragility problem, strong coupling and low cohesion problem, and incompleteness problem;
- a comprehensive analysis of the state-of-the-art concerning existing tools and solutions related to the three open problems;
- our vision of how the three problems hindering web test automation can be mitigated. Specifically, this concerns the development and adoption of a novel integrated testing environment called NEONATE;
- the overall description of the NEONATE integrated testing environment prototype, together with some usage scenarios.

The chapter is organized as follows: Section 2 introduces the three open problems in the context of web test automation, and Section 3 describes the pertinent literature. Section 4 illustrates our vision to overcome the three open problems, while Section 5 details the NEONATE integrated testing environment. At last, Section 6 reports two simulated usage examples of NEONATE pertaining to the creation and refactoring of a web test suite, and Section 7 describes the possible impact of NEONATE from both the perspectives of the researchers and practitioners. Section 8 concludes the chapter and outlines the future work.

2. THE THREE OPEN PROBLEMS IN THE CONTEXT OF E2E WEB TESTING

E2E testing is a type of black box testing based on the concept of *test scenario* (or test logic), a sequence of steps/actions performed on the web

application under test. One or more *test cases* can be derived from a test scenario by specifying the actual input data to use in each step and the expected outcomes. Test cases can be manually executed by a human on the browser or they can be implemented into *test code* (also known as test scripts). To this extent, testers use a high-level programming language (e.g., Java, Python, Ruby) to develop scripts that consist of commands simulating the user's actions on the GUI and retrieving information to be used in assertions verifying the expected outcomes.

The main benefits of adopting test automation are the possibility of (1) executing the test cases more often, for instance overnight, so that to increase the probability of bugs detection, (2) finding bugs on the early stages of development, and (3) reusing test code across successive releases of the web app under test (i.e., to catch regressions[a]).

However, despite these benefits, test automation is not "a silver bullet" and has limitations well known by testers. A thorough list can be found in [11], where the authors overview benefits and challenges of test automation based on empirical studies and experience reports in the industry. Among the outlined limitations, two are particularly severe when the web domain is considered: the *difficulty in maintenance of test automation artefacts* and the fact that *automation cannot replace manual testing*.

The first limitation can be better defined in two concrete problems namely, the *fragility problem* and the *strong coupling and low cohesion problem*.

The second limitation has deep roots and can be also associated with the *incompleteness problem*. In fact, not all testing tasks can be—or are worth to be—automated [8], thus developers automate only the test cases that they judge to be more crucial in their setting. As a consequence, an automated test suite covers only a small suboptimal portion of the application.

The authors of this work have matured similar opinions as Rafi's and colleagues by both working within industrial projects [9, 10] and performing academic empirical studies [12–14]. Other researchers have acknowledged the existence of these issues and have proposed initial solutions (see Section 3 for further details).

2.1 The Fragility Problem

The maintenance of test code during software evolution is the chief problem of web test automation, because the cost is expected to grow with: (1) the application size, (2) the number of test cases, and (3) the inevitable

[a] Regression testing aims at verifying that software previously developed and tested still behaves correctly even after the occurrence of an evolution or maintenance activity.

application evolution. Research works have singled out the manual maintenance of *locators* to be particularly problematic and expensive [13, 15]. Locators are specific commands used by test automation tools to identify the web elements on the GUI, before to perform actions on them. Examples of actions consist of clicking on a link, or filling in a text field in a form. Locators are used, for instance, to identify and fill the input portions of a web page (e.g., the form fields), to execute navigations (e.g., by locating and clicking on links) or to verify the correctness of the output (e.g., by locating the web page elements showing the result of a computation). It has been shown that even slight modifications of the application under test have a massive impact on locators [13]. For instance, changes as simple as renaming a page element or altering the choices in a dropdown list can cause locators to break (e.g., they become unable to select the desired element in the web page). Thus, the specific characteristics of the web applications make the test cases more *fragile*, rendering their maintenance extremely difficult and expensive as compared to maintaining test cases for a desktop application.

Let us clarify this problem by considering a simplified web application composed of two web pages—insertInfo.php and showInfo.php—that allow to insert and visualize personal information of the users. The test code for Version 1 of this web application (Test in Fig. 1) opens the insertInfo.php page, fills the form (shown in Fig. 1, top), submits the information and verifies that the inserted data are correctly displayed in the showInfo page (not shown for brevity). In this way, it is possible to test the correct insertion of the information in the application. To implement this test, it is necessary to locate some web page elements as, for instance, the field of the form for inserting the mobile phone number (i.e., the highlighted target element 1234 shown in Fig. 1). To this aim, a developer can use tools for the automatic generation of locators as FirePath,[b] which generates XPath expressions for the elements in a web page that can be used as locators. In our example, suppose that FirePath would produce the following XPath locator for the "Mobile" text field: //*[@id="userInfo"]/tr[3]/td[2].

We now consider a new version of the web application (Version 2). In this evolved version, the user is required to type also the gender information, and the corresponding text field has been inserted between the "Surname" and the "Mobile" text fields (see Fig. 1, bottom). In this scenario, `Test` is no longer able to select correctly the mobile phone number because its locator

[b] https://addons.mozilla.org/firefox/addon/firepath/.

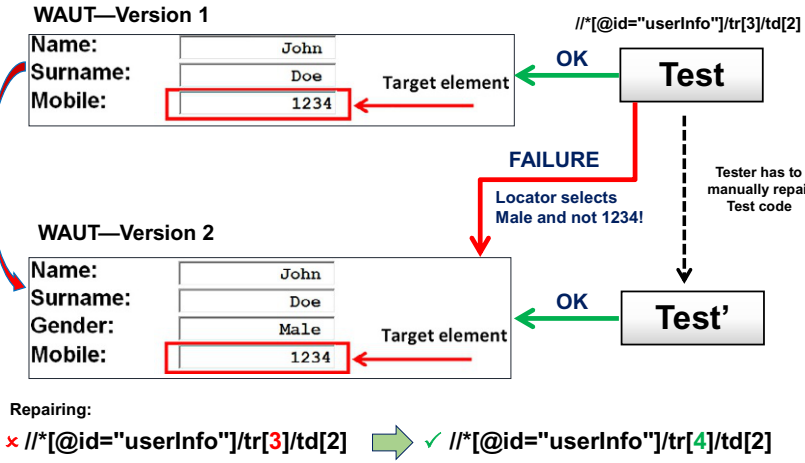


Fig. 1 Fragility problem explained by means of an example.

"points to" another element, i.e., the "Gender" text field. As a consequence, let us assume that the test would break, because the input validation of the application would not consider a phone number as a valid gender (e.g., the application could stop the insertion and visualize an error message). Hence, `Test` must be manually repaired.

In particular, the locator could be modified from //*[@id="userInfo"]/tr[3]/td[2] to //*[@id="userInfo"]/tr[4]/td[2]) in order to select the correct web element.

2.2 The Strong Coupling and Low Cohesion Problem

Test code produced by means of test automation tools often tends to merge two different notions: the test scenario (i.e., *what* to test) with test implementation (i.e., *how* to test). As a result, the test code is full of implementation details, that do not pertain to the test scenario. Thus, tests become difficult to read and understand, and costly to maintain and evolve. This is a common pitfall that leads to the strong coupling and low cohesion problem, i.e., the produced test code is *strongly coupled* with the web page structure and merges two different concerns. As an example, let us consider the fragment of code in Listing 1, which is a possible partial implementation of the `Test` of Fig. 1, developed with Selenium WebDriver [5]. The test code implements a portion of a simple test scenario which requires to enter the form data (name, surname, mobile number), click on the Enter button, and then (not shown) verify that the user has been correctly saved. As evident from our example,

the test code is strongly coupled with the web pages composing the application under test. The highlighted portions of the test are indeed related to technicalities such as locators used for retrieving the web elements to interact with (e.g., //*[@id="userInfo"]/tr[3]/td[2]), or command calls to the browser-specific APIs of the test automation tool (e.g., driver.findElement(...)) that fall outside the test scenario.

Listing 1 An excerpt of strongly coupled and low cohesive test code. Implementation details are highlighted in red.

```
@Test
public void testInsertInfo(){
  WebDriver driver = new FirefoxDriver();
  driver.get("localhost/webappundertest/insertInfo.php");
  driver.findElement(By.xpath("//*[@id='userInfo']/tr[1]/td[2]")).
      sendKeys("John");
  driver.findElement(By.xpath("//*[@id='userInfo']/tr[2]/td[2]")).
      sendKeys("Doe");
  driver.findElement(By.xpath("//*[@id='userInfo']/tr[3]/td[2]")).
      sendKeys("1234");
  driver.findElement(By.xpath("//*[@value='Enter']")).click();
  ...
  }
```

It is important to highlight that the strong coupling and low cohesion problem emphasizes the fragility problem discussed in Section 2.1. In fact, if the test suite is strongly coupled with the web application under test, several locators are inevitably repeated in the test code. If such locators become fragile after an evolution step, multiple repair activities would need to be carried out [14].

Fortunately, there is a solution to mitigate the strong coupling and low cohesion problem in the test code. The test scenario can be well separated from its technical implementation by using the Page Object (PO) design pattern[c] [16]. Page objects serve as an interface of the web app: they represent the GUIs as a series of object-oriented classes that encapsulate the features offered by each page into methods. Thanks to its adoption, the implementation details are moved into the page objects, a bridge between web pages and test cases, with the latter only containing the test logics.

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

2.3 The Incompleteness Problem

Even if techniques for automatically generating test cases for web applications have been proposed [17–19], typically test suites are still developed manually. Web testers study requirements documents and create test cases that cover the requirements of the web application under test. Then, they implement the test cases in test code. Being manual, these activities are time consuming, expensive, and not very effective. The direct consequence is the inability of the test suites to cover the input space of the application thoroughly, because only specific paths of the web application under test are exercised, leading to a low coverage of the functionalities and thus to a poorly tested web application.

We have observed this problem repeatedly during our industrial collaborations. For example, during a project with a company we found out that only a small portion of their tests was automated [10].

At last, we want to highlight that the fragility and strong coupling problems contribute to have incomplete test suites. Indeed, testers spend most of their time correcting/maintaining existing fragile/strong coupled code, rather than actually developing new test cases. In short, limiting the first two problems (fragility and strong coupling) implies to (indirectly) mitigate also the third one (incompleteness).

In conclusion, the three problems are correlated and, in order to find an effective solution, they need to be addressed *together*. We believe that by limiting the first two problems (fragility and strong coupling), also the incompleteness problem would be (indirectly) mitigated, leading to better tested web applications.

3. STATE OF THE ART ON THE THREE OPEN PROBLEMS

In the last 15 years, the research community has been particularly active in proposing new approaches and tools to advance the state of the art and practice in web test automation. Fig. 2 overviews the most relevant contributions and their mapping with the three open problems affecting web test automation and described in Section 2. In this section, we provide a detailed description of the investigations carried out by researchers and their findings. We, by no means, claim that our list represents all the relevant and noteworthy research performed in the area of web testing. However, we present the research solutions that, according to the experience and personal opinion of the authors, are mostly correlated with the three open problems.

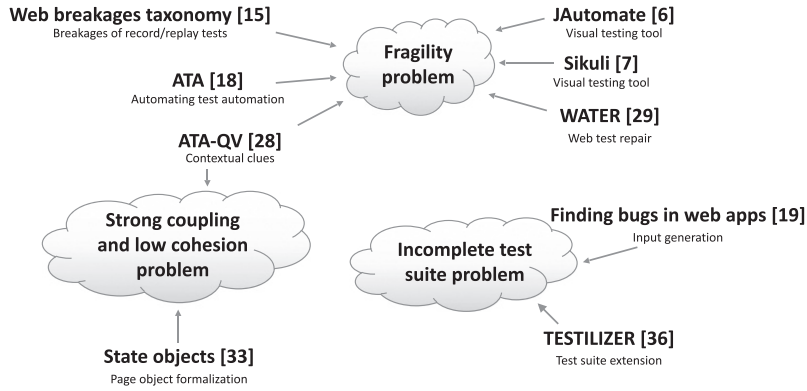


Fig. 2 Existing research proposals and tools addressing the three open problems individually.

3.1 State of the Art on the Fragility Problem

The brittleness of web test cases developed with test automation tools is a well-known problem among practitioners, and it has also been acknowledged and studied by many researchers [20–23]. However, a first study on the causes behind test breakages was by performed only recently, by Hammoudi and colleagues [15]. In this paper, they developed test suites for eight popular web applications and simulated a regression testing scenario, through the manual evolution of those test suites on 453 releases. Then, they characterized and collected the reasons for which tests broke. As an outcome, they developed a taxonomy of breakages for record/replay tests. As a confirmation of the findings of our previous research work [14, 24], also Hammoudi and colleagues have singled out *locators* as constructs that are particularly fragile in the face of software evolution, accounting for the three-quarters of the total amount of breakages.

The fragility of web test code can be mitigated in two different yet complementary ways. First, one can *prevent* the occurrence of test breakages, by creating test code which is designed to be *robust* to minor application changes. A second mitigation rule would be to use *repair* techniques to fix the broken test code automatically.

3.1.1 Robust Data Extraction

In the context of information retrieval and web data mining, researchers have proposed techniques for the robust extraction of information from evolving structured documents (e.g., XML documents). As an example, Dalvi and colleagues [25] propose to generate high-level XPaths that are

resilient to minor page changes and thus can be used to retrieve the same information in evolving versions of an HTML document. In another work [26] two models were used to study robustness: the adversarial model, which includes the worst-case robustness of wrappers, and the probabilistic model, which is based on the expected robustness of wrappers, as web pages evolve. By using both models, robust wrapper can be constructed. An evaluation on real websites demonstrated that such algorithms are highly effective in coping up with changes in websites and reduce the wrapper breakage by up to 500% over existing techniques.

The downside of such techniques is that they require learning a probabilistic model from a corpus of documents. For this reason, they cannot be directly adopted in a typical software engineering scenario, where the testing phase is usually characterized by strict timing constraints. In order to be useful, wrapper generation techniques must be adapted for generating robust locators—rather than wrappers—to be used in web testing environments, where there is a similar need by the tests to target the same web element, across different releases of the same web application.

3.1.2 Breakage Prevention

To this extent, the ROBULA+ algorithm [24] uses heuristics adapted from the information retrieval field in order to generate robust XPaths that can be used as locators in web test cases. A robust locator is likely to work correctly also if the web application undergoes minor GUI changes on new releases. The intuition behind ROBULA+ is to carefully combine XPath predicates in order to maintain the locators as short and compact as possible, while retaining a low level of fragility. Empirical results have demonstrated such an intuition: ROBULA+ locators exhibited a lower fragility than the state-of-the-practice/art locator generator algorithms such as FirePath or Selenium IDE.

LED [27] is a programming-by-example tool that automatically synthesizes web element locators based on positive and negative examples of DOM elements provided as input by the developer. LED casts the problem of finding a locator to solving a constraint satisfaction problem over the group of valid DOM states in a web application. Results show that LED can synthesize DOM element locators with a 98% recall and 92% precision, with as low as five positive and five negative relevant examples.

Other notable solutions are ATA [20] and its successor ATA-QV [28]. For brevity, we describe only the enhanced latest version ATA-QV. The tool uses visual landmarks (named contextual clues in the paper) to identify

objects on the GUI. For each element with which the test interacts, ATA-QV retrieves a list of potential visual "clues" to be used as reliable landmarks to identify such elements uniquely. Let us consider the form of Fig. 1. ATA-QV would automatically retrieve the labels that are associated to each text field (i.e., that are positioned above, or besides them), so that, for example, the first text field would be associated to its descriptive label "Name", the second text field with the label "Surname", and so on. By means of this mapping, ATA-QV then converts manual test steps into more abstract commands, which should be more resilient to the evolution of the applications, if the set of GUI changes remains limited. Even if very promising, ATA and ATA-QV suffer two major limitations: (1) if a descriptive label cannot be found, the tool will use the typical DOM-based locators such as the XPath of the element, making the script susceptible of the fragility problem, (2) the tool does not guarantee that the labels will be persistent across versions (for instance, if the tagname of the element gets modified—from `input` to `button`—then the tool would fail in targeting the correct element), and (3) such techniques are embedded into commercial tools, and are not available "as-is" to practitioners.

3.1.3 Web Test Repair

In the context of Web, the state-of-the-art test repair algorithm is WATER [29]. WATER is based on differential testing and compares the execution of the tests on two different releases: one in which the tests run properly, and another in which the tests break. By gathering data about these executions, WATER uses heuristics to find a set of potential fixes for the broken tests. While the repair algorithm of WATER has a straightforward design and can manage a good number of cases (such as locators or assertions), it has limitations that derive from its DOM-related narrowness. First, the algorithms can produce a great number of false positives, as recognized by the authors of the paper [29]. Second, only relying on the DOM may be insufficient to find candidate repairs. Third, the algorithm only triggers repairs at the point in which the test stops, which makes it impossible to handle propagated breakages [15], i.e., cases in which a breakage appears in a later point of the test execution. In many of such cases, it is imperative for the tester to inspect the GUI or replay the tests to find the root cause behind a breakage, because existing repair algorithms are inapplicable.

Recently, an enhanced repair algorithm has been proposed, named WATERFALL [30]. WATERFALL is built on top of WATER and suggests repairs

for broken Selenium IDE test scripts. The algorithm is also based on differential testing, but it does take into account all the intermediate minor versions that occur between two major releases of a web application and uses the WATER heuristics to trigger the repairs. The results of the empirical study on seven web applications show that WATERFALL is more effective than WATER (with a 209% improvement on the number of correct repairs suggested). This is because the number of code changes between two major releases is typically larger than the number of changes between any pair of successive releases or commits, and are also be intertwined in manners that render repair heuristics less effective. Applying a repair technique iteratively to intermediate versions or commits leaves the technique with fewer, less intertwined changes to consider per application, increasing its chances of success [30].

3.2 State of the Art on the Strong Coupling and Low Cohesion Problem

As far as maintenance is concerned, Page Object has proven successful in web test automation, where it has emerged as the leading pattern for enhancing test maintenance, reducing code duplication and lowering the coupling between test cases and web applications. Page objects provide an unique location for maintenance activities. Since no duplications are present, a single code fragment needs to be corrected, and the modification is propagated on the entire test suite.

Martin Fowler first described the page object pattern under the name Window Driver [16]. Fowler illustrates basic rules of thumb for page object creation, as that "*it should provide an easy programmable interface to the program, hiding the underlying widgetry in the window.*" Furthermore, Fowler advocates assertions-free page objects, meaning that "*although they are commonly used for testing, they should not make assertions themselves. Their responsibility is to provide access to the state of the underlying page, leaving the testers to carry out the assertion logic*" [16]. However, the term "page object" has been popularized by the Selenium web testing framework, which has become the generally used name. Selenium's wiki strongly encourages the use of page objects as a best practice and provides advices on how they should be implemented [31].

Berner and colleagues [8] report on observation and experiences made in a dozen of projects in the area of test automation. Both works push on the concepts of reuse and single responsibility, of which the page object pattern is an important candidate for the implementation of such best practices within a web test suite. In another empirical study [32], studies on the

prevalence of Selenium-based tests among open-source applications are presented. The most frequent subjects to change are quantitatively and qualitatively assessed, e.g., it has been found that 75% of web tests are no longer maintained after two/three releases due to impracticality to coevolve them together with the software under test. In their future work, the authors suggest the use of design pattern, such as the Page Object, to maintain the traceability between unit tests and web application portions.

The benefits associated with the adoption of the Page Object pattern in the maintenance of web test suites have been empirically measured, both within an industrial environment and in academia. A first empirical work shows that test suites developed with a programmable approach (e.g., Selenium WebDriver) exhibit higher benefits than those developed with a capture and replay method (e.g., Selenium IDE) [12]. In addition, maintaining a programmable test suite required less effort, thanks to the introduction of the Page Object design pattern, able to decouple the test logics from that of the application. This insight is confirmed by another investigation within a real world industrial setting [10]. Thanks to the Page Object, the maintenance effort is drastically reduced when the software evolves, thus justifying their initial implementation effort. Unfortunately, in case of big web applications, the creation of page objects can be a laborious task.

Concerning how page objects should be implemented, a state-based generalization, based on UML state charts, is proposed in [33]. For Van Deursen, a behavioral state machine of the web application can effectively guide a tester toward the generation of page objects, following the typical user scenarios. In his proposal, each page object corresponds to a state in the state machine, and hence becomes a *state object*. A state object has a set of well-defined responsibilities, defined by its methods. There are two kinds of behavioral methods. The *inspection methods* return the textual value of web elements displayed on the browser, when it is in a given state (typically, they can be used in test scenarios to verify that the browser displays the expected values, e.g., the user name of the current logged user must be present on the home page). The *trigger methods*, on the other hand, correspond to functions that make the browser change state. Inspection methods can also be useful as a self-check, to verify that a series of constraints hold when the application is in a particular state (for example, in the authenticating state, one would expect to find input fields for the insertion of user name and password).

In a recent work, Yu and colleagues [34] propose an automatic test generation technique for dynamic web applications. The technique decouples

test code from web pages by automatically generating page objects. Based on the page objects, tests are created performing an iterative feedback-directed random test generation. On top of the retrieved methods, sequences of calls are generated with Randoop [35] for the automatic construction of test cases.

3.3 State of the Art on the Incompleteness Problem

Achieving total coverage in testing is definitely impossible, due to the limited amount of resources or the combinatorial explosion of the inputs in case of complex applications. However, research efforts have been directed to improve the state of the art with test augmentation and test generation techniques.

To the best of our knowledge, TESTILIZER is the first work aimed at *extending* an existing web application test suite by leveraging existing test cases [36]. This approach reuses knowledge in existing human-written test cases and uses a web crawler to extend the navigation paths. In particular, TESTILIZER exploits input values in existing tests to explore alternative paths and mined oracles for regenerating assertions for such alternative paths. Results show that, on average, TESTILIZER can generate test suites with improvements of up to 150% on the fault detection rate and up to 30% on the code coverage, compared to the original test suite.

Current web testing techniques simplify the test oracle problem in the generated test cases by using "soft" oracles, such as invalid HTML and runtime exceptions [36]. However, "soft" oracles are limited and not able to find bugs pertaining to the functional requirements of the application. To be really useful, automatically generated test cases should contain "strong" oracles (i.e., assertions) to determine whether the application under test works as requested. Indeed, it has been empirically evaluated that assertions are strongly correlated with the test suite effectiveness [37]. Code coverage is certainly a desirable characteristic for a test suite. However, that paper shows that there is a very strong correlation between the effectiveness of a test suite and its size, with the quantity and quality of assertions it contains.

In the context of web test generation, ARTEMIS is a feedback-directed automated test generation for JavaScript in which execution is monitored to collect information that directs the test generator toward inputs that yield increased coverage [38]. The generated tests can be used for instance to detect HTML validity problems and other programming errors. Mesbah

and colleagues [17] propose a new testing technique that features the power of automated exploration (by means of a crawler) with invariant-based testing. In this technique, the user interface is checked against different constraints, expressed as invariants, which can act as oracles to automatically conduct sanity checks at a DOM level. For example, generic invariants include the absence of broken links, or the validity of the HTML. Concerning JavaScript applications, JSEFT is a framework that targets test generation for JavaScript applications. The approach employs a combination of function coverage maximization and function state abstraction algorithms to efficiently generate unit test cases with automatically generated mutation-based oracles [39]. ATRINA [40] infers test oracles from existing UI-level test cases to generate JavaScript unit tests. SUBWEB [41] is a new search-based web test generation technique, in which page objects and genetic operators are used to drive the generation of both test inputs and feasible navigation paths. On a first case study, SUBWEB was able to achieve higher coverage of the navigation model than a typical crawling-based approach.

4. OVERCOMING THE THREE OPEN PROBLEMS: THE NEONATE VISION

After the insights gained in years devoted to research in the web testing field [12, 23, 24, 42–46] and our analysis of the state of the art and practice related to this field, we have matured a vision of how to overcome the existing open problems affecting web test automation.

4.1 The Stuck Situation

First, we want to summarize the current problematic situation in three points as follows.

– First, web applications are complex, heterogeneous, distributed systems that are highly dynamic with unpredictable control flows. Specific requirements characterize them, such as huge time-to-market pressure, distributed infrastructure, high asynchronicity, and constant shifts in user requirements. Thus, in those years, developing high-quality web applications and testing web systems have become one of the most challenging goals among software engineers, demanding for novel approaches/techniques.

– Second, web testers are often not able to develop robust and maintainable test code. One might argue that this is due to the testers' inexperience or the lack of skills. While this is likely possible, we believe the increasing complexity of web applications and the three aforementioned problems (see Section 2) to have a predominant role.
– Third, when test code is produced, it often exhibits characteristics as fragility, strong coupling with the web application under test, and incompleteness. Thus, such automated test suites are underused or quickly abandoned, despite their potential value to catch errors and regressions. As a result of this tangled situation, we currently live with "buggy" web applications.

Thus, what we really need is a major breakthrough in web application testing, i.e., a new way of doing test automation.

4.2 The Vision

The vision behind the NEONATE project is to *empower the Web tester* with an *integrated testing environment* (ITE) composed by techniques and tools that are specifically designed to deal with (and hopefully overcome) the three big problems affecting web test automation.

In our vision, NEONATE will offer the tester automatic support in the: (1) creation of robust test code, (2) repair of broken test code, (3) separation of the test logic from the implementation details, (4) extension of existing test suites, and (5) migration to novel visual testing technologies.

We believe that NEONATE will facilitate the development and maintenance of web test suites. As a result, web applications will be tested more intensively, and hopefully this will benefit their correctness.

The need of integrating various testing tools in a ITE has been pointed out by several researchers in the last 20 years. For instance, one of the first works on this topic is the one of Gao et al. [47]. They developed a web-based system for test information sharing, control and management. Moreover, they report that big companies have strong demand for integrated testing environments providing: (1) test information bank (supporting the test information sharing among engineers in different phases), (2) facility tools for test information tracking, analysis and reporting, and (3) a systematic transparent interface to plug-in various test tools. Subsequently, Williams et al. in [48] describe an architecture for integrating both new and existing testing tools into solutions adopted in testing organizations across IBM. More in detail, this integrated testing environment supports the control and data integration

across tools and repositories, provides an unique graphical user interface, and can be extended to support new tools.

Agreed with the authors of these works, we designed the architecture of NEONATE on a plug-in infrastructure. In this way the functionalities offered by our ITE can be easily extended to include the most novel state-of-the-art solutions that will be eventually proposed in the future. Moreover, NEONATE is equipped with a GUI-based module that first analyses the test code looking for code deficiencies, and then presents—in a coherent, unified way—several wizards conceived to help the tester in executing the suggested improvement tasks (see additional details in Section 5).

4.3 Existing Integrated Testing Environments

To the best of our knowledge, no integrated testing environments (ITE) have been proposed to face simultaneously the three open problems in the context of web testing.

A notable ITE is FITTEST [49], whose purpose is to automatically generate test cases for web applications. Such test cases are generated using a combination of different techniques: model-based testing, combinatorial testing, mutation, and search-based testing techniques. NEONATE, on the other hand, encompasses a set of prototype tools for improving, maintaining, and extending a web test suite. Thus, the goals of the two projects are orthogonal and can be eventually utilized together (however, the engineering integration cost between the artifacts produced by two different ITEs might be nonnegligible).

FITE [50] is a static and dynamic analysis ITE, not specific for web applications. We share with the authors of FITE the same basic idea: developing an integrated testing environment able to continuously analyze the increments and produce recommendations to the tester. As in our proposal, the role of the user is essential, and the purpose of the testing environment is to empower the human tester. One of the most interesting aspects of FITE is its pluggable view-based approach, i.e., the tester can select a kind of analysis (e.g., performance analysis) and the tool produces only recommendations related to that selection. For example, for the "performance" selection, the tool might show code paths with the highest execution times. The main difference with respect to NEONATE is in the level of testing: FITE is able to assist the developer with unit tests whereas NEONATE manages E2E test code. Moreover, the considered quality factors are different: FITE targets security and performance aspects, while NEONATE focuses on the quality and correctness of the test source code.

One of the first works emphasizing the importance of having an ITE in the web application context is by Margaria and colleagues [51] where the authors present a methodology with the aim of granting coverage of all the functionalities of a web application. The core module is the Test Coordinator that drives the generation, execution, evaluation, and management of the system-level tests. Differently from NEONATE, in [51] a model checker and the concept of property are at the base of the proposal, as well as a graphical test cases design facility that is not present in NEONATE.

5. ARCHITECTURE OF THE NEONATE INTEGRATED TESTING ENVIRONMENT

In this section, we present the architecture of the NEONATE integrated testing environment. In our vision, NEONATE will be a full-fledged toolset composed by a set of joinable integrated prototypes (hereafter referred also as *modules*), each of which addresses a particular development/maintenance task on web test code. NEONATE will be built on top of Selenium WebDriver, the flagship open-source test automation tool for web applications. We have opted for the WebDriver framework because it is a remarkable open-source solution, widely adopted both in the academia and in the industry [32]. The maturity of WebDriver on the worldwide testing scenario has been also recognized by the W3C consortium, of which it has become a standard.[d] NEONATE will be realized by relying on the Eclipse platform and its plug-in-oriented architecture. Fig. 3 gives a high-level overview of NEONATE: each module will communicate by means of a lightweight protocol, will share a common repository containing the test code (repository software architecture), and will eventually be combined in a tool-chain. On top of the plug-in architecture, a GUI will allow the web tester to use the tools in stand-alone modality or combine them. NEONATE will support modules for the development of test code (i.e., those labeled with "D") and the maintenance of existing test code (i.e., those labeled with "M"). NEONATE will feature the ASSISTANT module, i.e., an orchestrator mechanism based on wizards able to suggest the testers which plug-in, or set of plug-ins, needs to be executed for a specific purpose. For instance, if the tester wants to refactor an existing Selenium DOM based web test suite, to enhance its robustness and structural quality, the ASSISTANT will suggest to apply in series the modules ROBULA+ and APORES. In the

[d] http://www.w3.org/TR/webdriver/.

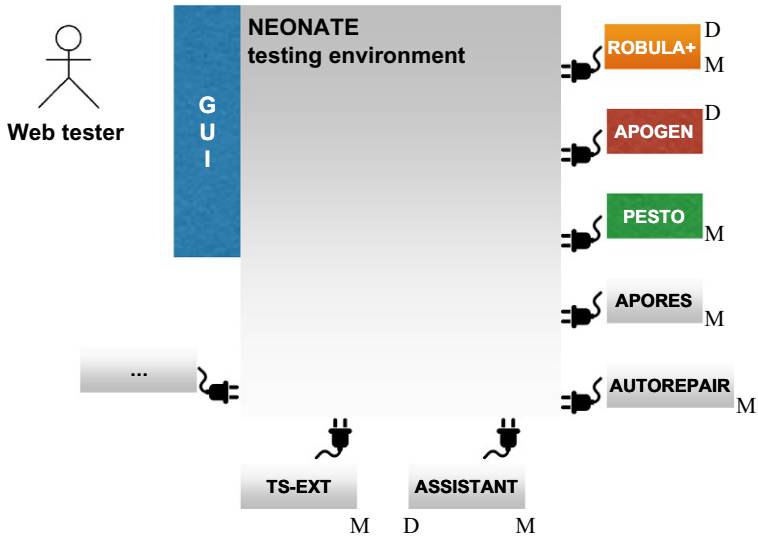


Fig. 3 High-level architecture of the NEONATE integrated testing environment.

following sections, we describe in turn each module of NEONATE. We underscore that some prototypes have already been implemented as part of our research (the colored modules in Fig. 3) while some others are part of our ongoing work (the gray modules).

5.1 Robust Web Element Locators With ROBULA+

Usually, the first aspect on which it is important to intervene is the test suite robustness. As already said, locators play a major role in the fragility problem. Thus, the first module of NEONATE integrates ROBULA+ (ROBUst Locator Algorithm+) [24], a state-of-the-art algorithm for automatically generating robust web element locators. We define a robust locator as a selection command that continues to point to the target web element, even if the web page has changed because of a new release of the web application. ROBULA+ is based on the XPath query language. The intuition behind ROBULA+ is to carefully combine XPath predicates in order to maintain the locators as short and smart as possible, while retaining a low level of fragility. In short, the algorithm starts with the most generic XPath locator that selects all nodes in the DOM tree (//*). It then iteratively refines the locator until only the element of interest is selected. In such iterative refinement, ROBULA+ applies seven refinement transformations, according to a set of heuristic XPath

specialization steps, prioritization, and black listing techniques. The prioritization is used to rank candidate XPath expressions in terms of expected robustness, while the black list excludes attributes that are intrinsically considered fragile. For further technical details, we refer the reader to [24].

We have demonstrated the effectiveness of ROBULA+ in producing reliable locators and limiting the fragility of test cases. In brief, we have compared the robustness of the locators generated by state-of-the-art/practice tools and algorithms (i.e., FirePath absolute and ID-relative, Selenium IDE, and Montoto [52]) with the ones generated by ROBULA+. Empirical results (see Fig. 4) indicate that the locators generated by ROBULA+ are significantly better in terms of robustness than all the other kinds of locators (63%–90% fragility reduction) which is expected to be associated with a corresponding reduction of the maintenance effort required to repair the test cases. Moreover, the time required by ROBULA+ for generating the XPath locators is acceptable for a human web tester (only 0.16 s per locator on average).

We have implemented ROBULA+ as a freely available Java program, able to generate (in batch mode) XPath locators for hundreds of web elements (useful, for instance, when the locators of an entire test suite have to be changed with the ones generated by ROBULA+). Moreover, ROBULA+ is also available as a Firefox add-on that can be used by web testers to generate locators during the development of test suites. We are also considering the idea of including in NEONATE the capability of combining the locators produced by a set of different algorithms (including ROBULA+) into a single, consolidated *multilocator* [23] based on a voting mechanism that assigns different voting weights to different locator generation algorithms. The two implementations of ROBULA+ are available on the tool's web site: http://sepl.dibris.unige.it/ROBULA.php.

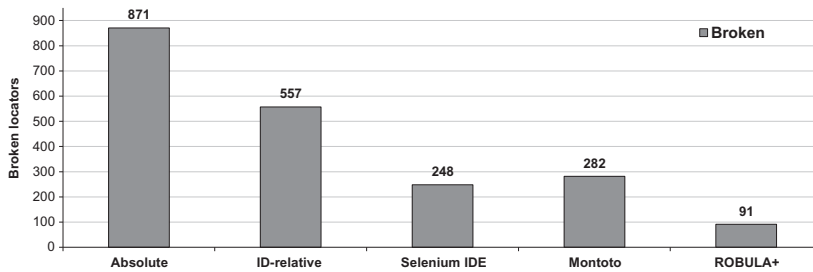


Fig. 4 Number of broken locators out of 1110 analyzed web elements in eight web applications (see [24]).

5.2 Automatic Generation of Page Objects With APOGEN

While developing a web test suite, it is of great importance to separate the test logic from its technical low-level implementation. With this aim in mind, the second module of NEONATE integrates APOGEN (Automatic Page Object GENerator) [53], a prototype tool for the *automatic generation of page objects* to support E2E web testing. Automatically generated page objects can alleviate the manual work of web testers so as to reduce the development costs.

APOGEN consists of five main modules (see Fig. 5): a Crawler, a Clusterer, a Cluster Visual Editor, a Static Analyser, and a Code Generator. The input of APOGEN is any web application, together with the input data necessary for the login and forms navigation. The output is a set of Java page objects as supported by the Selenium WebDriver framework. In short, APOGEN infers a model of the target web application by reverse engineering it by means of an event-based crawler (we chose Crawljax, a state-of-the-art tool for the automatic crawling of interactive web applications [17, 54]). Then, similar web pages are clustered into syntactically and semantically meaningful groups. The event-based model and the additional information (e.g., DOMs and clusters) are statically analyzed to generate a state object-based model. At last, this model is transformed into Java page objects, via model to text transformations. Since our ultimate goal is the automatic generation of meaningful page objects we share and tried to incorporate in the development of APOGEN the guidelines and best practices reported in [33].

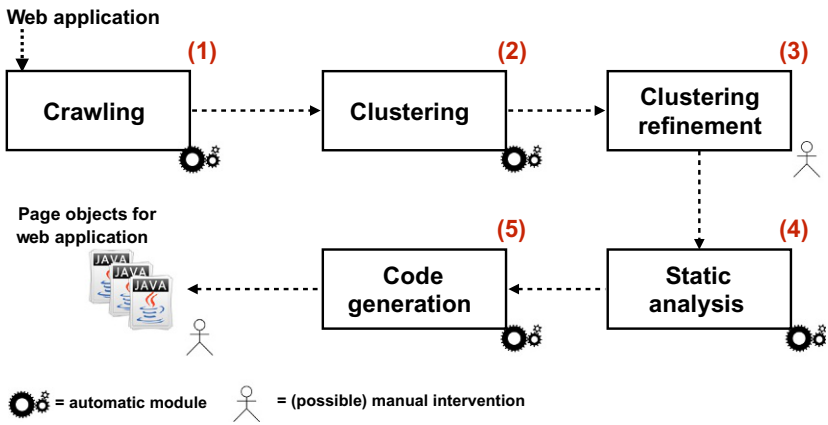


Fig. 5 High-level overview of APOGEN's approach for web page objects creation (see [53]).

Experimental results on six existing web applications indicate that APOGEN is effective to group semantically related web pages [55]. Furthermore, the page objects obtained from the output of clustering are very similar to the page objects that a developer would create manually. Indeed, 75% of the code generated by APOGEN can be used "as-is" by a Web tester, breaking down the manual effort for page object creation. Moreover, a big portion of the page object methods (84%) created to support assertion definition corresponds to meaningful and useful behavioral abstractions.

For further technical details, we refer the reader to [53]. All details about APOGEN, demo video, and experimental data can be found on the tool web site: http://sepl.dibris.unige.it/APOGEN.php.

5.3 Generating Visual Test Suites With PESTO

Depending on the characteristics of the web application under test, changing the approach used for localizing the web elements to interact with could change the fragility of the locators and thus the associated maintenance effort. More in detail, nowadays there are two main approaches to web element localization [13, 14, 56]. (1) the DOM-based approach (supported by, e.g., Selenium WebDriver[e]), where test cases access the web page Document Object Model (DOM) to locate the web elements (e.g., anchors, buttons) by accessing their properties (e.g., identifier or text), or by navigating the DOM tree by means of XPath queries. On the other hand, (2) the visual approach (adopted by, e.g., JAutomate [6] and Sikuli [7]) relies on image recognition techniques to identify and control GUI components.

PESTO (PagE object tranSformation TOol) [57, 58] is a tool able to transform a Selenium WebDriver test suite into a Sikuli visual test suite automatically, while retaining the same coverage and all the assertions.

PESTO can be useful when the web application under test is evolved to adopt modern visual widgets such as Google Maps. In these cases, DOM-based tools are not adequate, because the DOM of these visual components is complicated to retrieve (if not impossible), and such tools do not support visual testing (e.g., they cannot assert that an image is *visually* present on the screen). Moreover, with PESTO, companies can evaluate the benefits of third generation tools at minimum cost, without taking the risk of a substantial investment, necessary for the manual migration of existing test suites. Automatically migrated test cases can be smoothly introduced in the existing

[e] http://www.seleniumhq.org/projects/webdriver/.

testing process, so as to evaluate their effectiveness and robustness in comparison with the existing DOM-based suites.

To the best of our knowledge no other solution, both in the academia and in the industry, yet exists to carry out such migration task for Selenium-web based test cases. A proposal in the context of desktop applications is by Alégroth and colleagues [56], which allows to migrate existing automated component-based GUI test cases (GUITAR [59]) to the visual approach (VGT GUITAR).

PESTO relies on aspect-oriented programming, computer-vision, and code-transformations. PESTO executes the transformation by means of two main modules (see Fig. 6). The Visual Locators Generator generates a visual locator for each web element used by the DOM-based test suite (i.e., a unique image representing that web element on the web application GUI). Specifically, PESTO automatically retrieves the bounding rectangle of the web elements, and an image is automatically captured for each web element surrounded by such rectangles. This is technically realized by capturing the command calls to the web elements with aspect-oriented programming (AOP). Based on the captured images, the original test suites

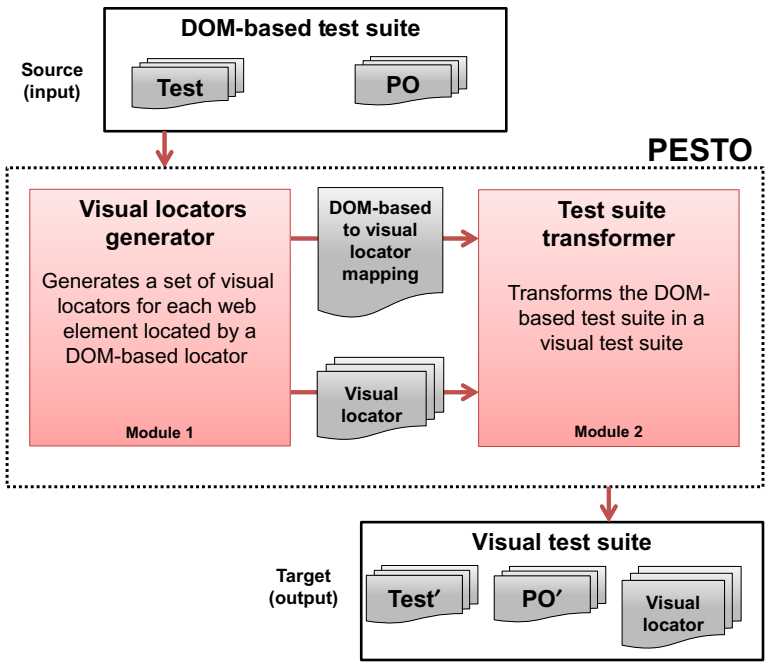


Fig. 6 High-level logical architecture of PESTO.

are automatically rewritten as visual test suites by the Visual Test Suite Transformer. This module converts the source code of the DOM-based test suite in order to adopt the visual approach; the majority of the changes are concentrated in the page objects code, since page objects are responsible for the interaction with the web pages.

The effectiveness of PESTO has been evaluated on a set of DOM-based validation test suites, developed by an independent, professional web tester for different open-source web applications [57]. PESTO generated the corresponding visual test suites for the DOM-based validation test suites requiring only a small amount of alignment work. The visual test suites automatically generated by PESTO were then executed and the test cases exhibited the correct, expected behavior. In our study, PESTO was able to migrate 100% of the command calls used in the existing DOM-based validation test suites. Moreover, by analyzing other existing DOM-based test suites, we found that PESTO can handle more than 95% of the employed command calls, when an abstraction as the page object is used. The visual locators automatically generated by PESTO were checked for readability by the professional tester involved in our experiments and they were judged easy to understand.

For the interested reader, a demo video of PESTO and the source code can be found on the tool web site: http://sepl.dibris.unige.it/PESTO.php.

5.4 Separating Test Specification from Test Implementation With APORES

The goal of the APORES (Automatic Page Objects REStructurer) prototype is separating test specification from test implementation in test code suffering the strong coupling and low cohesion problem. More concretely, APORES (which is one of the modules still to be implemented) will attack the challenging problem of restructuring a test suite built without the page objects into an equivalent one adopting them (see Fig. 7). While APOGEN creates page objects from scratch (thus supporting development), APORES will infer page objects from existing test code. Further, the two prototype tools can be used together by the web tester to alleviate the incompleteness problem: APORES will be used to transform a legacy test suite and APOGEN to generate fresh page objects that can be used to extend the initial test suite.

We are aware that the task is challenging to be executed in an automatic way. However, we are confident that a big portion of the transformation can be conducted automatically, with techniques similar to those of the already implemented prototypes of NEONATE. From the technical point of view,

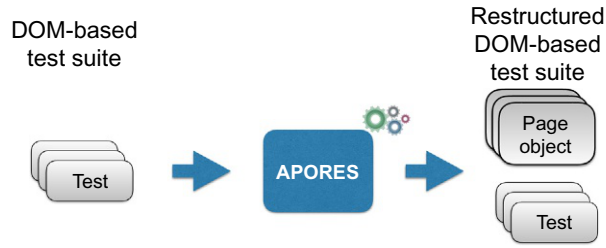


Fig. 7 Input and output of APORES: it is evident the presence of the novel created page objects.

APORES could re-use part of the AOP infrastructure fine-tuned for PESTO and the crawler used for APOGEN. Further, automatic code transformation techniques (e.g., using the TXL language [60]) will be employed to effectively execute the migration.

5.5 Suggesting and Executing Repairs for Broken Code With AUTOREPAIR

The goal of the AUTOREPAIR prototype is suggesting and eventually executing repairs for broken test code due to web application evolution. Automatic repairing of test suites is an alternative way to face the fragility problem. ROBULA+ would *prevent* and limit the amount of broken locators; on the other hand, AUTOREPAIR would intervene in all cases in which a *repair* is needed. In our vision, AUTOREPAIR would manage also breakages that go beyond locator problems, for instance, a breakage in the test workflow due to a missing statement. Self-repairing test suites is a big open research problem that goes beyond the web context (e.g., techniques could be adapted to work also in the mobile environment). Indeed, several research groups are currently working on the test suite self-repairing problem [23, 29, 30, 61, 62].

As a first step toward the automatic repair of web test suites, we intend to use differential testing to compare the test executions of two successive releases of a web application: a first in which the test suite runs properly, and a second in which breakages occur (as done in WATER [29]). Then, by means of static and dynamic analyses techniques, we aim at (1) automatically determine the root cause behind the breakage, (2) automatically determine repairs for the broken statements, (3) rank the candidate repairs according to some heuristics, (4) suggest the ordered list of repairs to the developer for inspection, and (5) automatically fix the test with the chosen repair.

This approach is simple but effective because it is based on the assumption that the evolution of the web application under test is limited. In the

comparison, we will use information contained in error stack traces and output messages produced by previously instrumented test code (as done for example in the ReAssert tool [61]). AUTOREPAIR would manage programmable WebDriver test code instead of capture and replay (C&R) test code (in the case of WATER specifically Selenium IDE test code) and we intend to use effective similarity measures [63] and prioritize the given suggestions to reduce the number of false positives repair suggestions.

In our vision, AUTOREPAIR will consider the test suites self-repairing problem in its entirety, also managing changes at level of steps of the test scenario (due to business logic changes) and input data (e.g., the format of the data is changed from DD/MM/YYYY to MM/DD/YYYY). In particular, repairing automatically the steps of a test scenario is extremely challenging. Indeed, repair operations can range from simple insertion and deletion of steps (e.g., deleting a confirmation action) to repair of multiple steps that require to execute multiple operations in series. Considering input data, one factor limiting the usage of web application test automation techniques is the cost of finding appropriate input values. To mitigate this problem, Elbaum and colleagues proposed a family of techniques based on user-session data [64]. In general, user-session-based techniques: (1) collect user interactions when users use a web application, (2) store the clients requests in the form of URLs and name-value pairs, and then (3) apply strategies to generate test cases. Several mechanisms can be used to capture and store user-session data (e.g., one of the simplest is configuring the web server to log all the received requests). An example of access log for an e-commerce application could be the following:

```
(2018-01-01 12:00:00) Client1 Home
(2018-01-01 12:00:01) Client1 SearchProd, prodCat=notebook
(2018-01-01 12:00:02) Client2 Home
(2018-01-01 12:00:03) Client2 FindShop, place=London,
country=UK
(2018-01-01 12:00:03) Client1 FindShop, place=Rome,
country=IT
(2018-01-01 12:00:04) Client3 Home
(2018-01-01 12:00:06) Client3 AdvSearchProd, pMin=200,
pMax=800
(2018-01-01 12:00:07) Client3 AdvSearchProd, date=01-01-
2018
```

In particular, time stamps are assumed to be associated with each entry of the log. Then a column contains the name of the host requesting a web page (e.g., Client1). The next column contains the name of the requested page followed by the name-value pairs provided to the web server (e.g., via GET requests). When requests coming from the same host are found within a proper time interval (which depends on the specific implementation of the user-session data storage mechanism), it is assumed that navigation from a previously accessed page to a new one is taking place. Otherwise, a direct request of a page is considered to occur. Finally, when a request from a host is not followed by any other request from the same host, this is interpreted as the termination of the navigation session. From this simple example it is possible to extract useful input data such as geographical locations (e.g., London, UK and Rome, IT), dates (e.g., 01-01-2018), ranges of price (e.g., from 200 to 800), and name of products (e.g., notebook). These user-session data, collected in previous releases can be then reused as input data in the current release. However, also user-session data suffer the evolution problem: session data may become invalid due to changes in the application. Input data repairing techniques [65] are able to alleviate this problem; in this way input data are able to survive the application evolution.

Finally, meaningful input generation is known to be a challenging task, due to the difficulty to find meaningful values for forms (interesting proposals exist [66–68]), and the correlation they may have (e.g., the zip code value and the country to which it refers).

5.6 Extending Existing Test Suites With Ts-EXT

Human-written test suites can be a gold mine, i.e., a valuable source of domain knowledge related to which interactions are more important to cover, which data have been used as inputs, and what elements on the page need to be asserted and how.

The goal of the Ts-Ext (Test Suite EXTender) prototype is to improve the effectiveness of the target test suite by extending it. With improving the effectiveness of the target test suite we mean: augmenting the coverage and fault finding capability of the test suite.

To implement Ts-Ext a web crawler is necessary, i.e., a program that automatically retrieves web pages of a target application and builds a web graph model where, in the simplest case, nodes are web pages and edges are hyperlinks between pages. There are three specific problems to address: (1) automated input generation, (2) paths selection (among the unbounded

number of behaviors) and (3) assertions generation. Similarly to Fard et al., who proposed the TESTILIZER tool [36], we believe that these problems can be faced by combining the manual approach with an automated one.

The idea is the following: by starting from an existing limited test suite, TS-EXT will extend it automatically adding more test cases so as to cover unexplored paths in the web graph model. More in detail, TS-EXT will: (1) mine existing test code to infer existing input data and assertions, (2) execute the test code to find already explored web application paths, and (3) expand the initial human-written test suite with other novel test cases. To produce these novel test cases a Web crawler will be employed to discover unexplored paths while input data and assertions will be computed by means of an input generator, leveraging available user-session data [64] (i.e., data gathered when users exercise a Web application, as seen in Section 5.5), and using a tool for generating assertions.

More in detail, assertions can be created by levering on dynamic detection of likely invariants techniques similar to the ones implemented in Daikon [69]. An invariant is a property that can be relied upon to be true during the execution of a program or a portion of it. Invariants can be used in assert statements, documentation, and formal specifications. Examples include: being constant ($x == a$), nonzero ($x != 0$), being in a range ($a <= x <= b$), linear relationships ($y = ax + b$), ordering ($x <= y$), functions from a library ($x = f(y)$), containment (x belongs y), sortedness (x is sorted). The assertion generator of TS-EXT can benefit from a Daikon-like automatic invariant detector. As an example of possible additional assertion, consider a simple online shop web application. As a first step, it is required to trace a large number of user sessions. The resulting log will include information concerning interesting values shown on the web pages (e.g., the number of products in the carts during the users' navigations) and the info about every action performed by the users (e.g., adding or removing products). By analyzing the execution traces, the invariant detector is able, for instance, to find the following invariant: the number of products in the cart of a user must be always equal to the number of products she/he added in—or removed from—the cart (e.g., if five products have been added and then two have been removed, an automatically generated invariant will check that the cart contains three products). This kind of invariants can be added *automatically* as assertions in different points of the test scripts (for instance every time a product is added or removed from the cart).

Fig. 8 exemplifies our idea. `Tc0` is the test code written by the Web tester. Its execution produces the path composed by blue nodes on the web graph

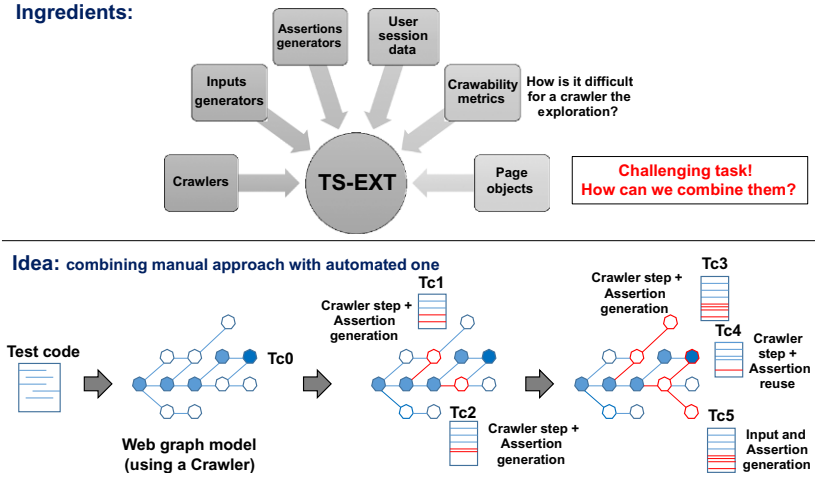


Fig. 8 Improving the effectiveness of a test suite with TS-EXT.

Testilizer assertion

```
1  AssertTrue(lastRow.getParent().getTag() == "table"
2        AND lastRow.getParent().getAttributes() == "list of table's attributes"
3        AND lastRow.getTag()=="tr"
4        AND lastRow.getAttributes()=="list of tr's attributes"
5        AND for each TD (X=1..n):
6              lastRow.getChild(X).getTag()=="td"
7              AND lastRow.getChild(X).getAttributes=="list of td X's attributes"
8  )
```

"*Human Like*" assertion

```
1  AssertTrue(table.getName=="John" AND  table.getSurname=="Doe" )
```

Fig. 9 An example of a TESTILIZER (*top*) and a *Human-like* (*bottom*) assertion.

model (see Fig. 8). On the contrary, red nodes (Fig. 8, center and right) represent further steps carried out by or via the aforementioned "ingredients" (e.g., input generator). `Tc1` …`Tc5` represent the code of test cases diversifying `Tc0`. For example, for generating `Tc1` it is sufficient to make two steps similarly to `Tc0` (i.e., the first two blue steps), a Crawler step and finally generating a fresh assertion by means of the generation assertion tool.

An ambitious goal would be to outperform TESTILIZER both in terms of effectiveness of the extended test suite (i.e., more coverage and more faults finding capability) and in terms of quality of the produced assertions. Indeed, the assertions produced by TESTILIZER are quite effective but difficult to understand and maintain because based on the structure of the page (see Fig. 9 top for an example). On the contrary, we intend to automatically produce "human-like" assertions, i.e., assertions simple to understand and more similar to those that would have produced a human (see Fig. 9 bottom for an example).

5.7 Supporting the Tester During Maintenance/Development With ASSISTANT

The ASSISTANT module will support the web tester in the use of NEONATE and its modules. The goal of the ASSISTANT is suggesting to the web tester the actions/transformations to apply to the test code that are most appropriate, in order to improve its quality, or to extend an initial test suite.

In our vision, the ASSISTANT is a recommendation system that will analyze the test code looking for code smells. A repository of *test code smells* containing potential problems identified in the test code during our analyses will be maintained and continuously updated, as new potential threats manifest. The test code manually written by the tester will be continuously analyzed by means of a static analyzer (*diagnosis phase*), starting from the earliest stages of development. The ASSISTANT will consult the test code smells repository and provide the tester with suggestions/recommendations. The suggestion can be: (1) *on demand*, i.e., explicitly requested by the tester or (2) *in automatic modality*, when the ASSISTANT will automatically warn the tester about a potential threat in the code, and suggest the NEONATE module that can be adopted to mitigate/solve it (e.g., ROBULA+, APORES, and TS-EXT). Thus, suggestions are displayed based on the task that the tester is performing on the test suite. For instance, if the tester is editing a test code using fragile absolute XPath locators in the editor, the ASSISTANT will automatically highlight the identified smell suggesting her/him the usage of the ROBULA+ module.

A dialog-based GUI will guide the tester through the steps necessary to execute the selected refactoring. Depending on the complexity of the refactoring (in some cases, multiple modules have to be called in series), a wizard or a simple dialog will be used to gather the necessary information. Similarly to the Eclipse refactoring wizard, the ASSISTANT will be equipped with a preview module, able to show the changes that the refactoring will perform. This will open a view containing a list of changes to be made to the current test code along with a link that allow to execute the refactoring on the test code.

6. NEONATE'S EXAMPLES OF USE

This section aims at showing how NEONATE can be useful in simplifying typical test code tasks. Two scenarios will be considered: (1) creation from scratch of an automated test suite for a web application and (2) refactoring of an existing automated test suite. Besides these two examples, NEONATE can be utilized for many other maintenance activities

such as: generating robust locators for new or modified web elements with ROBULA+, generating page objects for new portions of the web application with APOGEN, realigning broken test code with the updated web application with AUTOREPAIR.

6.1 Automated Test Suite Development (Scenario 1)

Let us assume that the quality assurance team (QAT) currently verifies the correct behavior of the web application employing several web testers that manually follow the steps of a predefined set of test cases. This is actually a typical solution adopted in the industrial practice, as observed in our experience. However, as discussed before, this practice is error prone, time consuming, and ultimately not very effective.

For this reason, the QAT wishes to improve the level of automation of the testing activities by means of automated testing tools. The desired goal is to bring all the main benefits of automation in the current inefficient testing setting, e.g., a fast and unattended execution of the test cases after every change made to the web application under test (i.e., for regression purposes).

Without a ITE like NEONATE, the QAT has to manually implement the test code able to instrument the web application and verify its behavior. QAT can follow two strategies [14]:

- adopt a capture and replay (C&R) tool (e.g., Selenium IDE). In this former case, the web testers perform actions on the web application GUI and execute the test scenario. The tool records such actions and generates replayable test scripts that, in a second time, can redeliver the same actions to the browser automatically.
- rely on a programmable test automation framework (e.g., Selenium WebDriver). In this latter case, testers have to implement the test code using a programming language (such as Java or Ruby) and APIs providing commands to control the browser and interact with the web elements composing the pages (e.g., click a button, fill in a field, or submit a form).

Both strategies have shown nonnegligible development/maintenance effort [12] that often represents a barrier to the adoption of test automation. More in detail, C&R test suites requires less development effort as compared to the programmable ones. On the other hand, programmable tools show, if the test code is well-engineered, a lower overall cost (i.e., considering both development and maintenance costs) since the adoption of specific design patterns. As already discussed, the page object pattern can drastically reduce the maintenance effort of a test suite during the web application evolution.

By adopting NEONATE, instead, the QAT can get all the benefits of high-quality test automation code at a cost comparable with the one of a C&R solution. In fact, the ASSISTANT contains a wizard procedure that suggests the steps to follow during the creation of a new test suite. The first step consists in developing high-quality page objects representing the web pages of the application. This step is fully automated in NEONATE. Indeed, the ASSISTANT asks only a few information about the target web application such as its URL and the data to insert in the forms that cannot be filled with autogenerated inputs (e.g., the login credentials). At this point, APOGEN is executed and a set of page objects encapsulating the web pages functionalities is created. Thanks to the integration with ROBULA+, the locators used in the page objects for locating the web page elements to interact with are as robust as the best locators that a human expert would create. Fig. 10 shows a fragment of a sample web application which is composed by various pages such as Login, Index, Messages, Settings, etc. For each of them, APOGEN generates a page object encapsulating their functionalities (reported as gray boxes in Fig. 10). For instance, the Login page contains a form allowing the users to

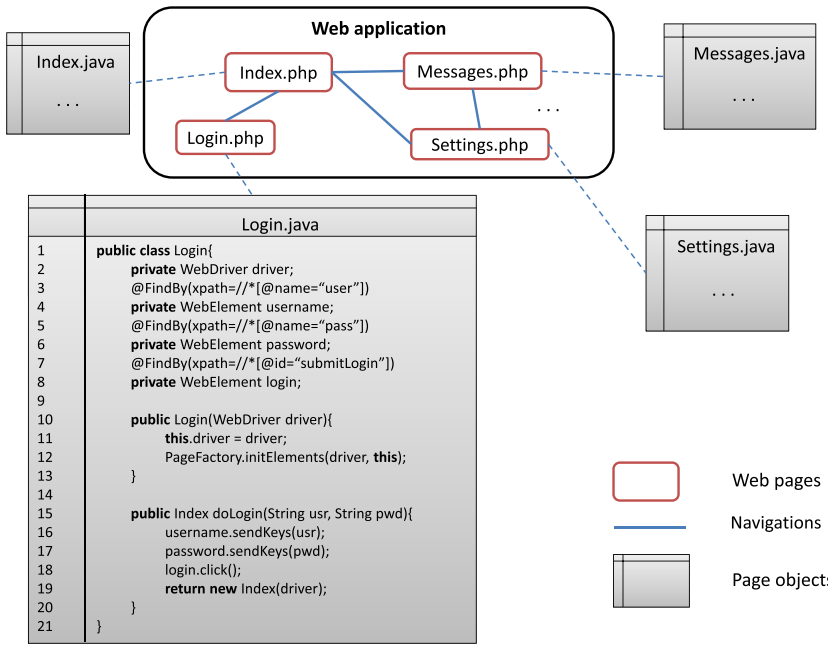


Fig. 10 Fragment of the web application and associated page objects created by APOGEN.

```
1  @Test
2  public void testLogin(){
3        Login lp = new Login(driver);
4        Index ip = lp.doLogin("admin", "secret");
5        assertEquals(ip.getLoggedUser(), "(admin)");
6  }
```

Fig. 11 Example of page object-based programmable automated test.

log in the application. Correspondingly, a Login page object is created containing a method that allows to perform the authentication (see lines 15–20).

In the next step, the QAT has to manually implement the test cases into test code by simply calling the various methods exposed by the page objects and following the tests specifications (before driving the manual testing activities). Fig. 11 reports an example of test code for verifying the correct behavior of the login functionality relying on the page objects created by APOGEN.

At this point, the automated test suite creation is completed and the test suite executes the same test cases that were previously executed in a manual fashion. In addition, the ASSISTANT suggests to improve the coverage of the test suite by using the TS-EXT module. In this way several additional test cases can be added to the original test suite. For instance, in our example, the test suite contains only a positive login test (i.e., log in using correct credentials). Thus, TS-EXT is able to find the different navigation paths that appear when inserting incorrect or incomplete credentials and leading to one or more "error" pages displaying specific messages to the user (e.g., invalid credentials, empty password). Hence, new test cases are created, increasing the coverage of the initial test suite.

Finally, whether the QAT has decided that it is better to adopt the visual localization strategy for the web elements to interact with (instead of the DOM-based one that is the default option in NEONATE), because for example a refactoring toward complex visual components to offer increased user-friendliness and responsiveness is forthcoming, it is possible to automatically migrate (a portion of) the freshly created DOM-based test suite to the visual approach by mean of the PESTO module.

6.2 Automated Test Suite Refactoring (Scenario 2)

In this second example of use, let us assume that the QAT has already adopted Selenium IDE as a reference tool for web testing. Fig. 12 shows a Selenium IDE test script used to verify the correct behavior of the login

#	Action	Locator	Value
1	open	localhost/webapp/login.php	
2	type	//*[@id='LoginForm']/input[1]	"admin"
3	type	//*[@id='LoginForm']/input[2]	"secret"
4	click	//input[@id='submitLogin']	
5	assertText	//*[@id='loggedUser']	"(admin)"

Fig. 12 Example of C&R automated test.

```
1  @Test
2  public void testLogin(){
3      WebDriver driver = new FirefoxDriver();
4      driver.get("localhost/webapp/login.php");
5      driver.findElement(By.xpath("//*[@id='LoginForm']/input[1]")).sendKeys("admin");
6      driver.findElement(By.xpath("//*[@id='LoginForm']/input[2]")).sendKeys("secret");
7      driver.findElement(By.xpath("//*[@id='submitLogin']")).click();
8      assertEquals (driver.findElement(By.xpath("//*[@id='loggedUser']")).getText(), "(admin)");
9      driver.quit();
10 }
```

Fig. 13 Example of programmable automated test not adopting page object and factory patterns.

functionality. However, QAT quickly discovered several limitations in its usage. For instance, a C&R tool does not provide natively some useful features, such as conditional statements, loops, logging functionality, exception handling, and parameterized (also known as data-driven) test cases. Moreover, as mentioned in the previous section, C&R solutions have been shown to be very expensive during maintenance. For this reason, QAT could plan to migrate the test suite to Selenium WebDriver, i.e., adopting a programmable approach.

Without NEONATE, the effort of such migration is, in practice, similar to developing a new programmable test suite from scratch. Indeed, the only support provided by Selenium IDE is to export the test cases in Java, one of the languages supported by Selenium WebDriver. However, in order to take full advantage of the benefits of the programmable approach it is required to provide the test code with a completely different organization with respect to the one created by the simple one-to-one translation from the Selenium IDE C&R test code. Indeed, the adoption of the Page Object and Factory patterns drastically affects the test suite structure. Fig. 13 shows the test code (written using the Selenium WebDriver framework), implementing the same test case, that does not adopt any design pattern (the same result could be obtained using the `export` functionality of Selenium IDE). This test code, even if small, is largely suboptimal considering

the two factors: robustness of the locators and structural quality. On the other hand, the test code reported in Fig. 11 is by far simpler and more understandable since all the technical details are moved to the page objects.

With NEONATE such onerous refactoring activities are executed automatically. Indeed, the ASSISTANT, analyzing the Java code exported by Selenium IDE, suggests the use of the APORES module for restructuring the test suite built without the page objects into an equivalent one adopting them. As reported in the previous section for APOGEN, also APORES takes advantage of ROBULA+ for generating robust web element locators. At this point, the programmable automated test suite is completed and executes the same test cases that were previously executed with Selenium IDE. As described in the previous section, following the ASSISTANT suggestions, it is then possible, for instance, to improve the test suite coverage by means of the TS-EXT module or migrate it to the visual approach with the PESTO module.

7. NEONATE'S LONG-TERM IMPACT

We believe that NEONATE will represent a major breakthrough in the web testing domain. Moreover, since web apps are similar to mobile apps many of our proposals and prototype tools can also be applied to the mobile context increasing the impact of our work. More in detail, we believe that the impact of NEONATE will be on three lines: scientific, practical, and industrial.

7.1 Scientific

From the scientific viewpoint, NEONATE will boost the web test automation research in novel directions such as generating robust web test code, suggesting improvements to test code, migrating existing state-of-the-practice test suites to novel approaches. The research of this project is strategic, because it aims at raising the quality of all daily used apps and is original, because only few researchers in the world are facing these problems and with solutions complementary to ours. Currently, there is scientific evidence that test automation is hindered by the fragility problem, but its exact nature is unknown and unaddressed. Moreover, it is not clear whether specific ad hoc remedies and shortcuts are adopted in the industrial practice. Finally, possible relationships between the problem and the available testing tools are totally unknown. An industrial survey will be useful to answer all these questions.

7.1.1 Benchmarks

The possibility to compare the results of our work with those of other competitors is of vital importance. In general, well-defined *benchmarks* allow researchers to empirically validate their proposals and compare them against the state-of-the-art solutions. For instance, examples of benchmarks used in testing research involve SIR [70], Defect4J [71], and SF100 [72]. A benchmark of this kind is absent in web testing, thus another follow up of our research concerns building a benchmark of web applications, along with test suites, bug reports, and bug fixes. Such a benchmark could be a reference for the web testing community once made publicly available. We intend to build benchmarks on which we will validate our proposals and compare them with state-of-the art algorithms and tools. These benchmarks will be a reference for the community and we will make them publicly available. Note that, currently, although really useful, benchmarks are not available in our community.

7.2 Practical

From a practical viewpoint, the causal implication of our project is the following: we aim at nullifying the impact of the fragility problem. This will lead to a breakthrough in the practical usage of test automation. Extending the usage of test automation will increase the coverage of tested apps (deeply tested apps) with the indirect consequence of improving the quality of all web apps. This will have a remarkable economic relevance and direct impact on the daily lives of the users. It is worth noting that although our prototype tools are dependent on the chosen testing platform (e.g., Selenium), they will be almost independent from the technology used for developing the web app under test (e.g., Ajax, Angular, JQuery) because E2E testing tools consider the app as a black box and focus only on GUI interactions. Moreover, our solutions can be easily adapted/generalized to any E2E testing tool and framework.

7.3 Industrial

Even if it is outside the scope of the project to conduct technology transfer from academia to industry, the postproject outcome may include the development of full-fledged industrial tools and the creation of high-tech start-ups. Big companies and nonprofit organizations could integrate our solutions (e.g., the algorithms for generating locators currently embedded in the browsers produce locators that are extremely fragile) into their browsers as plug-ins or into specialized IDE for web developers and testers.

8. CONCLUSIONS

Nowadays web applications are critical assets of our society. Billions of people use them every day as source of information, means of communication and venue for commerce. For these reasons, checking their quality and correctness is of undeniable importance. Despite the advances in software testing, i.e., one of the possible ways to improve web applications quality, the ever-increasing technological complexity of modern web applications makes current testing techniques not adequate.

To face this issue, test automation tools are adopted by software engineers to automate the creation and execution of the test cases. However, such tools in industrial practice often leads to create test code which is (1) fragile to minor application changes, (2) strongly coupled with the web application under test and thus poorly maintainable, and (3) incomplete, i.e., having a low coverage of the web application input and functionalities. As a consequence, automated test suites are often abandoned, in spite of their potential value to catch errors and regressions.

In this work, we have discussed the three open problems that, according to our experience in web testing [12, 23, 24, 42–46] and our analysis of the state of the art and practice of this field, are among the major causes that limit the adoption of test automation in the web domain: fragility problem, strong coupling and low cohesion problem and incompleteness problem. We have then analyzed the existing research solutions to these problems, and we have presented our vision to overcome them, which concerns the development and usage of a new integrated testing environment called NEONATE. Finally, we have sketched our ongoing work on the NEONATE project which has the utmost purpose of representing a major breakthrough in the web testing domain.

As seen in the chapter, NEONATE, when completed, will be a powerful and flexible toolset focused on the development/maintenance/evolution of web test code. Currently three out of seven expected modules (i.e., ROBULA+, PESTO, and APOGEN) have already been implemented and the corresponding empirical evaluation shown that they outperform the state-of-the-art proposals. Future work will be devoted to the design, implementation, and validation of the remaining prototype tools and to empirically show the effectiveness of NEONATE as a whole testing framework.

In conclusion, we strongly believe that the NEONATE long-term project will demonstrate to the research community and practitioners that it is

possible to create robust test code, to repair broken test code, to increase the maintainability of test code, and, in the end, to improve the effectiveness of existing test suites.

In our opinion, web technologies will further increase their relevance in the next years. That is, everyday we use a web interface to access and use our applications and data. For this reason, we hope that this work could help the testing community to be more aware about the problems hindering web test automation and foster more researchers to find solutions. In fact, in the web context, new technologies emerge continuously and, each of them could require specific solutions to solve the three problems. For example, if a new web technology comes into the scene, the impact of the fragility problem might be more or less severe depending on the intrinsic characteristics of technology itself and that of the tool utilized for testing it (e.g., DOM-based or visual). NEONATE has been specifically designed to empower the web tester and limiting the three open problems. Thus, it has a human-centric vision aimed at supporting the tester activities. On other hand, novel computer-centric testing solutions based on artificial intelligence and machine learning are emerging with the aim of automatically creating and maintaining the test code. Their goal is to replace, as much as possible, the humans. We believe that the two approaches, human and computer centric, could (and probably should) coexist and as to reinforce each other with the overall goal to improve the final quality of the future web applications.

Moreover, also mobile applications are today very important for a great variety of activities and businesses; and this will be even more true in the next decade. NEONATE can influence closely related contexts such as mobile and IoT testing research. Especially for the former, we expect mobile apps testers to face similar problems as in the web context. Thus, we believe that our testing framework can be adapted also to solve issues affecting the mobile testing environment and novel solutions drew on the research presented in this chapter. As an example, the Android applications have XML-based interfaces that are conceptually similar to the HTML of the web applications. For this reason, similar challenges/research problems may concern robust localization of GUI elements, mobile page objects creation, visual image recognition, or hybrid test suites. Thus, it is realistic to hypothesize that the NEONATE ITE could be ported or implemented also to support mobile applications (for instance, by leveraging existing mobile testing frameworks and tools like Selendroid[f] or Appium[g]).

[f] http://selendroid.io/.
[g] http://appium.io/.

REFERENCES

[1] I.V. Yakovlev, Web 2.0: is it evolutionary or revolutionary? IT Prof. 9 (6) (2007) 43–45. ISSN: 1520-9202. https://doi.org/10.1109/MITP.2007.123.

[2] T. O'Reilly, What is web 2.0? Design patterns and business models for the next generation of software, 2005. http://oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html.

[3] A. Mesbah, Software analysis for the web: achievements and prospects, in: Proceedings of 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, IEEE, 2016, pp. 91–103. https://doi.org/10.1109/SANER.2016.109.

[4] H.S. Chaini, S.K. Pradhan, Test script execution and effective result analysis in hybrid test automation framework, in: Proceedings of International Conference on Advances in Computer Engineering and Applications, IEEE, 2015, pp. 214–217. https://doi.org/10.1109/ICACEA.2015.7164698.

[5] SeleniumHQ Web Browser Automation, 2017. http://www.seleniumhq.org/.

[6] E. Alégroth, M. Nass, H.H. Olsson, JAutomate: a tool for system- and acceptance-test automation, in: Proceedings of Sixth International Conference on Software Testing, Verification and Validation, ICST 2013, ISBN: 978-0-7695-4968-2, pp. 439–446.

[7] T.-H. Chang, T. Yeh, R.C. Miller, GUI testing using computer vision, in: Proceedings of 28th Conference on Human Factors in Computing Systems, CHI 2010, ACM, 2010, pp. 1535–1544.

[8] S. Berner, R. Weber, R.K. Keller, Observations and lessons learned from automated testing, in: Proceedings of 27th International Conference on Software Engineering, ICSE 2005, IEEE, 2005, pp. 571–579.

[9] M. Leotta, D. Clerissi, F. Ricca, C. Spadaro, Comparing the maintainability of Selenium WebDriver test suites employing different locators: a case study, in: Proceedings of First International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation, JAMAICA 2013, ACM, 2013, ISBN: 978-1-4503-2161-7, pp. 53–58. https://doi.org/10.1145/2489280.2489284.

[10] M. Leotta, D. Clerissi, F. Ricca, C. Spadaro, Improving test suites maintainability with the page object pattern: an industrial case study, in: Proceedings of Sixth International Conference on Software Testing, Verification and Validation Workshop, ICSTW 2013, IEEE, 2013, ISBN: 978-1-4799-1324-4, pp. 108–113. https://doi.org/10.1109/ICSTW.2013.19.

[11] D.M. Rafi, K.R.K. Moses, K. Petersen, M.V. Mäntylä, Benefits and limitations of automated software testing: systematic literature review and practitioner survey, in: Proceedings of Seventh International Workshop on Automation of Software Test, AST 2012, IEEE, 2012, ISBN: 978-1-4673-1822-8, pp. 36–42. http://dl.acm.org/citation.cfm?id=2663608.2663616.

[12] M. Leotta, D. Clerissi, F. Ricca, P. Tonella, Capture-replay vs programmable web testing: an empirical assessment during test case evolution, in: Proceedings of 20th Working Conference on Reverse Engineering, WCRE 2013, IEEE, 2013, ISBN: 978-1-4799-2931-3, pp. 272–281. https://doi.org/10.1109/WCRE.2013.6671302.

[13] M. Leotta, D. Clerissi, F. Ricca, P. Tonella, Visual vs DOM-based web locators: an empirical study, in: S. Casteleyn, G. Rossi, M. Winckler (Eds.), LNCS, Proceedings of 14th International Conference on Web Engineering (ICWE 2014), vol. 8541, Springer, 2014, pp. 322–340. https://doi.org/10.1007/978-3-319-08245-5_19.

[14] M. Leotta, D. Clerissi, F. Ricca, P. Tonella, Approaches and tools for automated end-to-end web testing. Adv. Comput. 101 (2016) 193–237. ISSN: 0065-2458. https://doi.org/10.1016/bs.adcom.2015.11.007.

[15] M. Hammoudi, G. Rothermel, P. Tonella, Why do record/replay tests of web applications break? in: Proceedings of Ninth International Conference on Software Testing, Verification and Validation, ICST 2016, IEEE, 2016, pp. 180–190.

[16] M. Fowler, Page Object, 2013. http://martinfowler.com/bliki/PageObject.html.
[17] A. Mesbah, A. van Deursen, D. Roest, Invariant-based automatic testing of modern web applications, IEEE Trans. Softw. Eng. 38 (1) (2012) 35–53. http://doi.ieeecomputersociety.org/10.1109/TSE.2011.28.
[18] S. Thummalapenta, K.V. Lakshmi, S. Sinha, N. Sinha, S. Chandra, Guided test generation for web applications, in: Proceedings of 35th International Conference on Software Engineering, ICSE 2013, IEEE, 2013, ISBN: 978-1-4673-3076-3. pp. 162–171. http://dl.acm.org/citation.cfm?id=2486788.2486810.
[19] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, M.D. Ernst, Finding bugs in dynamic web applications, in: Proceedings of International Symposium on Software Testing and Analysis, ISSTA 2008, ACM, 2008, ISBN: 978-1-60558-050-0, pp. 261–272. https://doi.org/10.1145/1390630.1390662.
[20] S. Thummalapenta, S. Sinha, N. Singhania, S. Chandra, Automating test automation, in: Proceedings of 34th International Conference on Software Engineering, ICSE 2012, IEEE, 2012, pp. 881–891.
[21] F. Ricca, M. Leotta, A. Stocco, D. Clerissi, P. Tonella, Web testware evolution, in: Proceedings of 15th International Symposium on Web Systems Evolution, WSE 2013, IEEE, 2013, ISBN: 978-1-4799-1608-5, pp. 39–44. https://doi.org/10.1109/WSE.2013.6642415.
[22] M. Leotta, A. Stocco, F. Ricca, P. Tonella, Reducing web test cases aging by means of robust XPath locators. in: Proceedings of 25th International Symposium on Software Reliability Engineering Workshops, ISSREW, IEEE, 2014, pp. 449–454. https://doi.org/10.1109/ISSREW.2014.17.
[23] M. Leotta, A. Stocco, F. Ricca, P. Tonella, Using multi-locators to increase the robustness of web test cases, in: Proceedings of Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, IEEE, 2015, ISBN: 978-1-4799-7125-1, pp. 1–10. https://doi.org/10.1109/ICST.2015.7102611.
[24] M. Leotta, A. Stocco, F. Ricca, P. Tonella, ROBULA+: an algorithm for generating robust XPath locators for web testing, J. Softw. Evol. Process 28 (3) (2016) 177–204. ISSN: 2047-7481. https://doi.org/10.1002/smr.1771.
[25] N. Dalvi, P. Bohannon, F. Sha, Robust web extraction: an approach based on a probabilistic tree-edit model, in: Proceedings of International Conference on Management of Data, SIGMOD 2009, ACM, 2009, ISBN: 978-1-60558-551-2, pp. 335–348. https://doi.org/10.1145/1559845.1559882.
[26] A. Parameswaran, N. Dalvi, H. Garcia-Molina, R. Rastogi, Optimal Schemes for Robust Web Extraction, vol. 4, VLDB Endowment, 2011.
[27] K. Bajaj, K. Pattabiraman, A. Mesbah, Synthesizing web element locators, in: Proceedings of 30th International Conference on Automated Software Engineering, ASE 2011, IEEE, 2015, pp. 331–341. https://doi.org/10.1109/ASE.2015.23.
[28] R. Yandrapally, S. Thummalapenta, S. Sinha, S. Chandra, Robust test automation using contextual clues, in: Proceedings of 25th International Symposium on Software Testing and Analysis, ISSTA 2014, ACM, 2014, ISBN: 978-1-4503-2645-2, pp. 304–314. https://doi.org/10.1145/2610384.2610390.
[29] S.R. Choudhary, D. Zhao, H. Versee, A. Orso, WATER: web application test repair, in: Proceedings of First International Workshop on End-to-End Test Script Engineering, ETSE 2011, ACM, 2011, ISBN: 978-1-4503-0808-3, pp. 24–29.
[30] M. Hammoudi, G. Rothermel, A. Stocco, WATERFALL: an incremental approach for repairing record-replay tests of web applications, in: Proceedings of 24th International Symposium on the Foundations of Software Engineering, FSE 2016, ACM, 2016, pp. 751–762.
[31] S. Stewart, Page Objects–Selenium wiki, 2017 https://github.com/SeleniumHQ/selenium/wiki/PageObjects.

[32] L. Christophe, R. Stevens, C.D. Roover, W.D. Meuter, Prevalence and maintenance of automated functional tests for web applications, in: Proceedings of 30th International Conference on Software Maintenance and Evolution, ICSME 2014, IEEE, 2014.
[33] A. van Deursen, Testing web applications with state objects. Commun. ACM 58 (8) (2015) 36–43. ISSN: 0001-0782. https://doi.org/10.1145/2755501.
[34] B. Yu, L. Ma, C. Zhang, Incremental web application testing using page object, in: Proceedings of Third Workshop on Hot Topics in Web Systems and Technologies, HOTWEB 2015, 2015, ISBN: 978-1-4673-9688-2, pp. 1–6.
[35] C. Pacheco, S.K. Lahiri, M.D. Ernst, T. Ball, Feedback-directed random test generation, in: Proceedings of 29th International Conference on Software Engineering, ICSE 2007, IEEE, 2007, pp. 75–84.
[36] A. Milani Fard, M. Mirzaaghaei, A. Mesbah, Leveraging existing tests in automated test generation for web applications, in: Proceedings of 29th International Conference on Automated Software Engineering, ASE 2014, ACM, 2014, ISBN: 978-1-4503-3013-8, pp. 67–78. https://doi.org/10.1145/2642937.2642991.
[37] Y. Zhang, A. Mesbah, Assertions are strongly correlated with test suite effectiveness, in: Proceedings of 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, ACM, 2015, ISBN: 978-1-4503-3675-8, pp. 214–224. https://doi.org/10.1145/2786805.2786858.
[38] S. Artzi, J. Dolby, S.H. Jensen, A. Møller, F. Tip, A framework for automated testing of javascript web applications, in: Proceedings of 33rd International Conference on Software Engineering, ICSE 2011, ACM, 2011, ISBN: 978-1-4503-0445-0, pp. 571–580. https://doi.org/10.1145/1985793.1985871.
[39] S. Mirshokraie, A. Mesbah, K. Pattabiraman, PYTHIA: Generating test cases with oracles for JavaScript applications, in: Proceedings of 28th International Conference on Automated Software Engineering, ASE 2013, IEEE, 2013, pp. 610–615. https://doi.org/10.1109/ASE.2013.6693121.
[40] S. Mirshokraie, A. Mesbah, K. Pattabiraman, Atrina: inferring unit oracles from GUI test cases, in: Proceedings of International Conference on Software Testing, Verification, and Validation, ICST 2016, IEEE, 2016, pp. 330–340.
[41] M. Biagiola, F. Ricca, P. Tonella, Search based path and input data generation for web application testing, in: Proceedings of Ninth International Symposium on Search Based Software Engineering, SSBSE 2017, Springer, 2017, pp. 18–32. https://doi.org/10.1007/978-3-319-66299-2_2.
[42] F. Ricca, P. Tonella, Analysis and testing of web applications, in: Proceedings of 23rd International Conference on Software Engineering, ICSE 2001, IEEE, 2001, pp. 25–34. https://doi.org/10.1109/ICSE.2001.919078.
[43] P. Tonella, F. Ricca, Statistical testing of web applications, J. Softw. Maint. 16 (1–2) (2004) 103–127.
[44] F. Ricca, P. Tonella, Detecting anomaly and failure in web applications, IEEE MultiMedia 13 (2) (2006) 44–51.
[45] A. Marchetto, F. Ricca, P. Tonella, A case study-based comparison of web testing techniques applied to AJAX web applications. Int. J. Softw. Tools Technol. Transfer 10 (6) (2008) 477–492. ISSN: 1433-2787. https://doi.org/10.1007/s10009-008-0086-x.
[46] A. Marchetto, P. Tonella, F. Ricca, State-based testing of Ajax web applications, in: Proceedings of First International Conference on Software Testing, Verification and Validation, ICST 2008, IEEE, 2008, pp. 121–130.
[47] J. Gao, C. Chen, Y. Toyoshima, D.K. Leung, Developing an integrated testing environment using the world wide web technology, in: Proceedings of the 21st International Computer Software and Applications Conference, COMPSAC 1997, IEEE, Washington, DC, USA, 1997, ISBN: 0-8186-8105-5, pp. 594–601. http://dl.acm.org/citation.cfm?id=645979.675991.

[48] C. Williams, H. Sluiman, D. Pitcher, M. Slavescu, J. Spratley, M. Brodhun, J. McLean, C. Rankin, K. Rosengren, The STCL test tools architecture, IBM Syst. J. 41 (1) (2002) 74–88. ISSN: 0018-8670.
[49] T.E.J. Vos, P. Tonella, J. Wegener, M. Harman, W. Prasetya, E. Puoskari, Y. Nir-Buchbinder, Future internet testing with FITTEST, in: Proceedings of 15th European Conference on Software Maintenance and Reengineering, CSMR 2011, IEEE, ISSN 1534-5351, 2011, pp. 355–358. https://doi.org/10.1109/CSMR.2011.51.
[50] M.W. Whalen, P. Godefroid, L. Mariani, A. Polini, N. Tillmann, W. Visser, FITE: future integrated testing environment, in: Proceedings of Workshop on Future of Software Engineering Research, FoSER 2010, ACM, 2010, ISBN: 978-1-4503-0427-6, pp. 401–406. https://doi.org/10.1145/1882362.1882444.
[51] T. Margaria, O. Niese, B. Steffen, Demonstration of an automated integrated test environment for web-based applications, in: Proceedings of Ninth International SPIN Workshop, LNCS, vol. 2318, Springer, 2002, pp. 250–253.
[52] P. Montoto, A. Pan, J. Raposo, F. Bellas, J. Lopez, Automated browsing in AJAX websites, Data Knowl. Eng. 70 (3) (2011) 269–283. ISSN: 0169-023X. https://doi.org/10.1016/j.datak.2010.12.001.
[53] A. Stocco, M. Leotta, F. Ricca, P. Tonella, APOGEN: automatic page object generator for web testing, Softw. Qual. J. 25 (3) (2017) 1007–1039. ISSN: 1573-1367. https://doi.org/10.1007/s11219-016-9331-9.
[54] A. Mesbah, A. van Deursen, S. Lenselink, Crawling Ajax-based web applications through dynamic analysis of user interface state changes, ACM Trans. Web 6 (1) (2012) 3:1–3:30.
[55] A. Stocco, M. Leotta, F. Ricca, P. Tonella, Clustering-aided page object generation for web testing, in: Proceedings of 16th International Conference on Web Engineering (ICWE 2016), LNCS, vol. 9671, Springer, 2016, ISBN: 978-3-319-38790-1, pp. 132–151. https://doi.org/10.1007/978-3-319-38791-8_8.
[56] E. Alégroth, Z. Gao, R. Oliveira, A. Memon, Conceptualization and evaluation of component-based testing unified with visual GUI testing: an empirical study, in: Proceedings of Eighth International Conference on Software Testing, Verification and Validation, ICST 2015, 2015, pp. 1–10. https://doi.org/10.1109/ICST.2015.7102584.
[57] M. Leotta, A. Stocco, F. Ricca, P. Tonella, PESTO: automated migration of DOM-based web tests towards the visual approach. J. Softw. Test. Verification Reliab 28 (4) (2018) e1665. ISSN: 1099-1689. https://doi.org/10.1002/stvr.1665 (John Wiley & Sons).
[58] M. Leotta, A. Stocco, F. Ricca, P. Tonella, Automated generation of visual web tests from DOM-based web tests, in: Proceedings of 30th Symposium on Applied Computing, SAC 2015, 2015, ISBN: 978-1-4503-3196-8, pp. 775–782.
[59] B.N. Nguyen, B. Robbins, I. Banerjee, A. Memon, GUITAR: an innovative tool for automated testing of GUI-driven software, Autom. Softw. Eng. 21 (1) (2014) 65–105. ISSN: 1573-7535. https://doi.org/10.1007/s10515-013-0128-9.
[60] J.R. Cordy, TXL—a language for programming language tools and applications, in: Proceedings of Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA 2004), Electronic notes in theoretical computer science, ISSN 1571-0661, vol. 110, Elsevier, 2004, pp. 3–31. https://doi.org/10.1016/j.entcs.2004.11.006.
[61] B. Daniel, D. Dig, T. Gvero, V. Jagannath, J. Jiaa, D. Mitchell, J. Nogiec, S.H. Tan, D. Marinov, ReAssert: a tool for repairing broken unit tests, in: Proceedings of 33rd International Conference on Software Engineering, ICSE 2011, IEEE, ISSN 0270-5257, 2011, pp. 1010–1012. https://doi.org/10.1145/1985793.1985978.
[62] M. Mirzaaghaei, F. Pastore, M. Pezzè, Automatic test case evolution, J. Softw. Test. Verification and Reliab. 24 (5) (2014) 386–411. ISSN: 0960-0833.

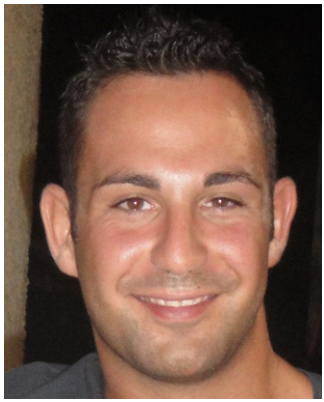
[63] S. Barman, S. Chasins, R. Bodik, S. Gulwani, Ringer: web automation by demonstration, in: Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, ACM, 2016, ISBN: 978-1-4503-4444-9. pp. 748–764. https://doi.org/10.1145/2983990.2984020.
[64] S. Elbaum, G. Rothermel, S. Karre, M. Fisher II, Leveraging user-session data to support web application testing, IEEE Trans. Softw. Eng. 31 (3) (2005) 187–202. ISSN: 0098-5589.
[65] M. Harman, N. Alshahwan, Automated session data repair for web application regression testing, in: Proceedings of the International Conference on Software Testing, Verification, and Validation, ICST 2008, 2008, ISBN: 978-0-7695-3127-4, pp. 298–307.
[66] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, Z. Su, Dynamic test input generation for web applications, in: Proceedings of International Symposium on Software Testing and Analysis, ISSTA 2008, ACM, 2008, ISBN: 978-1-60558-050-0, pp. 249–260. https://doi.org/10.1145/1390630.1390661.
[67] L. Mariani, M. Pezzè, O. Riganelli, M. Santoro, Link: exploiting the web of data to generate test inputs, in: Proceedings of International Symposium on Software Testing and Analysis, ISSTA 2014, ACM, 2014, ISBN: 978-1-4503-2645-2, pp. 373–384. https://doi.org/10.1145/2610384.2610397.
[68] J. Lin, F. Wang, Using semantic similarity in crawling-based web application testing, in: Proceedings of 10th International Conference on Software Testing, Verification and Validation, ICST 2017, 2017.
[69] M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, C. Xiao, The Daikon system for dynamic detection of likely invariants, Sci. Comput. Program. 69 (1–3) (2007) 35–45, ISSN: 0167-6423.
[70] H. Do, S. Elbaum, G. Rothermel, Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact, Empir. Softw. Eng. 10 (4) (2005) 405–435. ISSN: 1573-7616. https://doi.org/10.1007/s10664-005-3861-2.
[71] R. Just, D. Jalali, M.D. Ernst, Defects4J: a database of existing faults to enable controlled testing studies for java programs, in: Proceedings of 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, ACM, 2014, ISBN: 978-1-4503-2645-2, pp. 437–440. https://doi.org/10.1145/2610384.2628055.
[72] G. Fraser, A. Arcuri, Sound empirical evidence in software testing, in: Proceedings of 34th International Conference on Software Engineering, ICSE 2012, IEEE, 2012, ISBN: 978-1-4673-1067-3, pp. 178–188.

ABOUT THE AUTHORS

Filippo Ricca is an associate professor at the University of Genova, Italy. He received his PhD degree in Computer Science from the same University, in 2003, with the thesis "Analysis, Testing and Re-structuring of Web Applications". In 2011 he was awarded the ICSE 2001 MIP (Most Influential Paper) award, for his paper: "Analysis and Testing of Web Applications". He is author or coauthor of more than 100 research papers published in international journals and conferences/workshops. Filippo Ricca was Program

Chair of CSMR/WCRE 2014, CSMR 2013, ICPC 2011, and WSE 2008. His current research interests include: Software modeling, Reverse engineering, Empirical studies in Software Engineering, Web applications and Software Testing.

Maurizio Leotta is a researcher at the University of Genova, Italy. He received his PhD degree in Computer Science from the same University, in 2015, with the thesis "Automated Web Testing: Analysis and Maintenance Effort Reduction". He is author or coauthor of more than 60 research papers published in international journals and conferences/workshops. His current research interests are in software engineering, with a particular focus on the following themes: Web\Mobile\IoT application testing, functional test automation, empirical software engineering, business process modelling and model-driven software engineering.



Andrea Stocco is a postdoctoral fellow at the department of Electrical and Computer Engineering (ECE) of the University of British Columbia, Canada. He received his PhD in Computer Science at the University of Genova, Italy, in 2017, with the thesis "Automatic page object generation to support E2E testing of web applications". He is the recipient of the Best Student Paper Award at the 16th International Conference on Web Engineering (ICWE 2016). His research interests include web testing and empirical software engineering, with particular emphasis on test breakage detection and automatic repair, robustness and maintainability of test suites for web applications.