Search Based Path and Input Data Generation for Web Application Testing

3 authors:

Matteo Biagiola
Fondazione Bruno Kessler
5 PUBLICATIONS   **13** CITATIONS

SEE PROFILE

Filippo Ricca
Università degli Studi di Genova
177 PUBLICATIONS   **3,763** CITATIONS

SEE PROFILE

Paolo Tonella
Fondazione Bruno Kessler
294 PUBLICATIONS   **7,731** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

ASPIRE View project

Timbre research: physical and perceptual relationships among sounds View project

Search Based Path and Input Data Generation for Web Application Testing

Matteo Biagiola[1,2], Filippo Ricca[2], and Paolo Tonella[1]

[1] Fondazione Bruno Kessler, Trento, Italy
`biagiola,tonella@fbk.eu`
[2] University of Genova
`filippo.ricca@unige.it`

Abstract. Test case generation for web applications aims at ensuring full coverage of the navigation structure. Existing approaches resort to crawling and manual/random input generation, with or without a preliminary construction of the navigation model. However, crawlers might be unable to reach some parts of the web application and random input generation might not receive enough guidance to produce the inputs needed to cover a given path. In this paper, we take advantage of the navigation structure implicitly specified by developers when they write the page objects used for web testing and we define a novel set of genetic operators that support the joint generation of test inputs and feasible navigation paths. On a case study, our tool SUBWEB was able to achieve higher coverage of the navigation model than crawling based approaches, thanks to its intrinsic ability of generating inputs for feasible paths and of discarding likely infeasible paths.

Keywords: web testing, test case generation

1 Introduction

The main goal of end-to-end test case generation when the program under test is a web application is to ensure that the functionalities of the web application are fully exercised by the generated test cases. Usually no explicit navigation graph is available to guide the creation of test cases and to measure the degree of navigation coverage achieved. Existing approaches resort to web crawling in order to build the missing navigation model [1]. However, crawling is severely limited in its ability to fully explore the navigation graph, which depends on the input generation strategy. Such strategy is usually manual input definition, random input generation, or a mixture of the two. Another limitation of crawling based approaches is that not all paths in the crawled model are feasible (i.e., admit a test input that traverses them upon execution). As a consequence not all test paths derived from the crawled model can be turned into test cases that traverse the desired paths upon execution. When they don't cover the test paths for which they are generated, we say the test case is *divergent* (e.g., a step in a test case triggers an error, hence preventing the execution of the next steps; the app state does not allow the next action in the test path to be taken).

We address the problem of navigation graph construction by taking advantage of a design pattern commonly used in web testing: the Page Object (PO) design pattern. The purpose of POs is to encapsulate the details necessary to access the web elements in a web page (e.g., CSS or XPath locators, text extraction from HTML page, etc.) and to expose an abstract interface to developers, who can use it to perform high level operations against the web application under test. Such operations are exposed as methods of the PO class (e.g., a method to login; another to select an item and add it to a cart; etc.). The key benefit of this design pattern is the confinement of fragile web page access operations (e.g., implementation details concerning locators) within a single class, the PO, instead of spreading them across all test cases. This ensures higher maintainability of the test code when the web application evolves [2]. There is however another indirect benefit: when defining the POs for a web application, developers implicitly define also its navigation structure, since navigation methods in POs return the next PO encountered after triggering the navigation action. We resort to such property of POs to build the navigation graph to be covered by the automatically generated test cases.

We address the *problem of path feasibility* by means of a *novel search based test generation algorithm* (in particular, we use a genetic algorithm), which performs path selection and input generation at the same time. In our algorithm, a chromosome is a variable length sequence of navigation method invocations, each with the actual parameter values required for the invocations. Chromosomes are evolved by means of custom genetic operators that ensure compliance of the navigation sequences with the navigation graph and reachability of the remaining coverage targets. The fitness function that guides the generation of test cases is based on the distance between the executed sequence and the current set of coverage targets.

We have implemented our approach in the tool SUBWEB (Search based web test generator) and we have evaluated its effectiveness on the AddressBook case study, by comparing it with crawling based approaches. SUBWEB achieves higher coverage with smaller test suites than crawling based approaches. Moreover, the test cases generated by SUBWEB are feasible by construction, while on Address-Book the test cases derived from the crawled model are divergent test cases 17% of the times.

2   Related work

Several (semi-)automated web testing techniques have been proposed in the literature in the last few years, to reduce the human effort and the amount of work required for test case creation [3,4,1]. Most approaches rely on web application crawling for the construction of a navigation model and on graph visit algorithms for the selection of navigation paths that ensure high coverage of the model (e.g., transition coverage). Input data generation to turn the selected paths into executable test cases is either manual or random [5]. The proposal by Mesbah et al. [1] belongs to this category. They use a crawler, *Crawljax*,

to derive a state flow graph consisting of states and transitions that model the Ajax web application under test. Then, the tool *Atusa* uses the inferred model to generate test cases with predefined invariants as test oracles. Another approach for testing Ajax web applications has been proposed by Marchetto et. al. [4]. A Finite State Machine (FSM) that models the Ajax web application is built using a combination of dynamic and static analysis. Differently from *Atusa*, the adopted coverage criterion, used also in GUI-testing, is based on the notion of semantically interacting events. An alternative to *Atusa* is *Artemis*, a framework for automated testing of JavaScript web applications proposed by Artzi et al. [3]. The distinctive feature of *Artemis* is the usage of feedback-directed random testing [6].

There are several remarkable differences between our tool, SUBWEB, and existing approaches. First, coverage of the navigation model and input data generation are handled jointly by SUBWEB. Existing approaches [4,1] first generate a navigation model and then extract paths from it, without considering the problem that the generation of inputs for such paths might be difficult, requiring manual intervention, or even impossible, if the selected paths are infeasible. Another key difference is that input generation is search-based in SUBWEB, while it is either manual or random in existing approaches [3,4,1]. Finally, the abstraction from HTML pages to equivalence classes of pages that deserve separate test generation resorts to heuristic in existing approaches [4,1], while SUBWEB takes advantage of the abstraction defined by the developers when writing the POs for the web application under test.

To the best of our knowledge, the only attempt to use POs for test case generation is the proposal contained in a workshop paper by Yu et al. [7]. Similarly to *Artemis*, the proposed tool, called *InwertGen*, performs iterative feedback directed random test generation using the tool *Randoop* [6]. The key difference from our tool is that SUBWEB makes explicit use of the navigation model defined by developers through POs and uses a search-based approach, instead of a random one, to generate inputs that ensure high coverage of the navigation model.

3 Navigation Model Specification via Page Objects

POs are widely used in web testing to decouple the implementation details that depend on the web page structure from the test logics. The PO design pattern was first proposed by Martin Fowler[3] as an abstraction of the page under test that can be reused across test cases [2]. In fact, different test cases can refer to the same page object for locating and activating the HTML elements of the page under test, without having to duplicate the HTML access instructions multiple times, in different test cases.

While the main purpose of POs is to improve the modularization of the test code, POs implicitly specify a navigation model for the web application under

---

[3] https://martinfowler.com/bliki/PageObject.html

test. In fact, one of the best practices recommended for PO creation requires that PO navigation methods return the PO of the next page upon invocation [8]. This means that POs specify the navigation structure of the web application under test in terms of method invocations (i.e., operations executed within each abstract web page) and page objects returned by the invoked methods (i.e., next PO reached during navigation in the web application). We use such implicit navigation model for automated test case generation.

3.1   Page objects

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

```
1  public class ProductsPage implements PageObject {
2     public WebDriver driver;
3     public ProductsPage(WebDriver driver) {...}
4     public int getActiveCategory() {...}
5
6     public ProductDetailPage selectProduct(int id, int category) {
7        if((id >= 1 && id <= 6) &&
8          (category >= 1 && category <= 3) &&
9          (this.getActiveCategory() == category)) {
10          this.driver.findElement(By.id("product-" + id + "-" +
                 category)).click();
11          return new ProductDetailPage(this.driver);
12       } else {
13          throw new IllegalArgumentException("Invalid parameter values");
14       }
15    }
16 }
```

Fig. 1: PO example

Let us consider an example of e-commerce single page web application, named *Shopping Cart*. Figure 1 shows the code of the PO `ProductsPage`

Among others, this PO contains method `selectProduct` that models the user action consisting of the selection of a specific product from the product list displayed in the home page. The actual selection is performed at line 10 (if the precondition at lines 7-9 is satisfied), where Selenium WebDriver's APIs are used to locate and operate some web elements inside the concrete HTML page of the web application. Specifically, the web element of interest is located by its unique identifier, by means of the Selenium method `findElement(By.id(...))`. The action performed on the web element located by id is a click (Selenium method `click`, still at line 10). Since after the click navigation continues on the next page, which

is modelled by the PO `ProductDetailPage`, method `selectProduct` returns a new instance of the PO reached after the click, of type `ProductDetailPage`.

In general, PO methods may return values of any type (*void, int, String*, etc.). However, a recommended best practice is that *navigational* PO methods return the next PO encountered in the navigation (`this` if navigation does not leave the current PO). We strictly require that the tester specifies the navigation among the pages of the application through the POs returned by navigation methods, since we rely on them for the construction of the PO navigation graph. In the following, we call a *navigational method* any PO method that returns a PO. The second assumption that our technique makes on the way POs are written is that navigational methods include preconditions, i.e., each navigational method should specify the condition under which it can be safely executed. Such condition may depend on the invocation parameter values, as well as the state of the application, which is determined by the actions performed on the application in the previous navigation steps. In Figure 1, the precondition of method `selectProduct` deals with the proper selection of a product from the list of products shown in the home page. Each product is uniquely identified by the pair of parameters `id` and `category`. In the running example, the number of products shown in `ProductsPage` is known statically (it is always 6, for each category of products), while the category is the currently active category. So, the valid value for the `category` parameter must match the value returned by method `getActiveCategory`, while `id` can vary from 1 to 6. If the precondition is not respected, an exception is thrown.

We think the assumptions we make on how POs should be written to be processable by our technique are reasonable and do not impact to a significant extent the normal way in which developers write POs. In fact, the requirement that every navigational method returns the next PO is a best practice which is commonly followed, although it is not enforced by the PO pattern. The inclusion of preconditions is a bit more impactful, since in practice developers write test code that respect preconditions by construction, making them not strictly necessary. We think however that preconditions are a good programming practice, independently of the use of our technique. Moreover, in our experience (see Section 5), when they have to be written from scratch, such activity does not require much effort from the developers. In some cases it would be even possible to extract them automatically from the web application code (e.g., when parameter ranges can be obtained by static code analysis).

3.2 Navigation Graph

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

Definition 1 (PO Navigation Graph). *Given a set of page objects P, the associated navigation graph $G = \langle N, E \rangle$ consists of a set of nodes N bijectively mapped to P by function $po : N \to P$ and of a set of edges E that connect pairs of nodes $\langle n, m \rangle$ such that the page object $po(n)$ contains a* `return` *statement whose returned type is $po(m)$.*

Algorithm 1: Navigation graph extraction

---

1 **Procedure** extractNavGraph(*G, po*)

    Input:

    G: navigation graph computed so far

    po: page object to be analyzed

    Output:

    G: updated navigation graph

2   **begin**

3        $n := \text{getNodeByPO}(G, po)$

4        $l := \text{getNextPOsByStaticAnalysis}(po)$

5        $v = \emptyset$

6   **for** $po' \in l$ **do**

7            $m := \text{getNodeByPO}(G, po')$

8   **if** $m = NULL$ **then**

9                $m := \text{newNode}(po')$

10                $G.N := G.N \cup \{m\}$

11                $v := v \cup \{m\}$

12            $G.E := G.E \cup \{\langle n, m \rangle\}$

13   **for** $m \in v$ **do**

14            $\text{extractNavGraph}(G, \text{mapNodeToPO}(m))$

Algorithm 1 shows the recursive navigation graph extraction procedure. The loop at lines 6–12 iterates over all POs that are possibly returned by the PO under analysis. The set of such POs is obtained by static code analysis (line 4). When the returned PO is not already mapped to a graph node, a new node is created (line 9) and added to the graph (line 10). An edge $\langle n, m \rangle$ from the node n associated with the PO under analysis to the returned PO node m is then added to the graph (line 12). Graph extraction continues recursively on all newly created PO nodes (stored in variable v), i.e., all PO nodes not already present in the initial graph G (lines 13–14).

4   Search Based Path and Input Data Generation

Given the navigation graph $G = \langle N, E \rangle$, we can extract or generate test paths that exercise significant parts of the application. For instance, according to the *transition coverage* adequacy criterion, all edges E must be traversed at least once by the test paths. Formally, we can define a test path in the navigation graph as $p = \langle ns, es, pr \rangle$, where $ns \in N^+$ is a sequence of one or more graph

nodes; $es \in E^*$ is a sequence of zero or more edges, such that $\mid es \mid = \mid ns \mid -1$ and if $e_i = \langle n, m \rangle \in es$, then $n_i = n \in ns, n_{i+1} = m \in ns$; $pr \in V^*$ is a sequence of zero or more parameter names, equal to the parameter values required by the method invocations associated with es.

Let us take a simple path $p = \langle ns, es, pr \rangle$ from the *Shopping Cart* running example, with $ns = \langle$`ProductsPage, ProductDetailPage`$\rangle$, $es = \langle$`selectProduct`$\rangle$, $pr = \langle$`id, category`$\rangle$. The precondition of method `selectProduct` (see Figure 1) constrains the valid ranges of parameters `id` and `category` . As a consequence, not any arbitrary pair of integer values assigned to `id` and `category` will execute the path of interest. More generally, given a path p and a parameter sequence pr, we say p is *feasible* if there exists a parameter-value assignment that executes path p; we say path p is *infeasible* if there does not exist any parameter-value assignment that executes it. In order for a path $p = \langle ns, es, pr \rangle$ to be feasible, the conjunction of the constraints in the method preconditions associated with the edge sequence es must be satisfiable. Since some of the values evaluated in the method preconditions may depend on the server/client side state (e.g., `this.getActiveCategory()` in Figure 1), in general the problem of determining whether a path p is feasible or not is an *undecidable* problem. Moreover, since feasibility depends on the server/client state, which is computed by arbitrarily complex programs, SAT solvers are generally not a viable tool to address the path feasibility problem. For these reasons, we resort to a meta-heuristic algorithm. The *test generation problem* that we address (for the transition coverage adequacy criterion) is then to *generate a set of feasible paths, as well as the related parameter-value assignment, which, upon execution, ensure that all navigation graph edges are traversed at least once.*

4.1   Problem Reformulation

The problem of generating test cases that cover all navigation graph edges can be reformulated as a standard branch coverage problem on an *artificial class* generated from the navigation graph and the POs. In fact, a path $p = \langle ns, es, pr \rangle$ consists of a method sequence (namely, the sequence of method invocations associated with es), for which suitable parameter values must be found. Hence, we can solve the feasible path generation problem and the parameter input value generation problem by applying the search based approaches that have been proposed for object oriented testing [9], where method sequence and parameter values are generated at the same time. This requires the creation of an artificial class under test `CUT` whose methods are the methods associated with the navigation graph edges and whose state is the currently visited web page and more specifically, the currently instantiated PO for such web page.

Figure 2 shows the program transformation that creates class `CUT`. Its input is a set of POs and its output is class `CUT`, containing a private field to store the current page object, `cp`. Each PO method becomes a method of the new class, whose return type becomes `void`. The method can be called only if current PO `cp` is an instance of the PO where the method originally belonged to. When this condition is satisfied, current page object is cast to its concrete type and
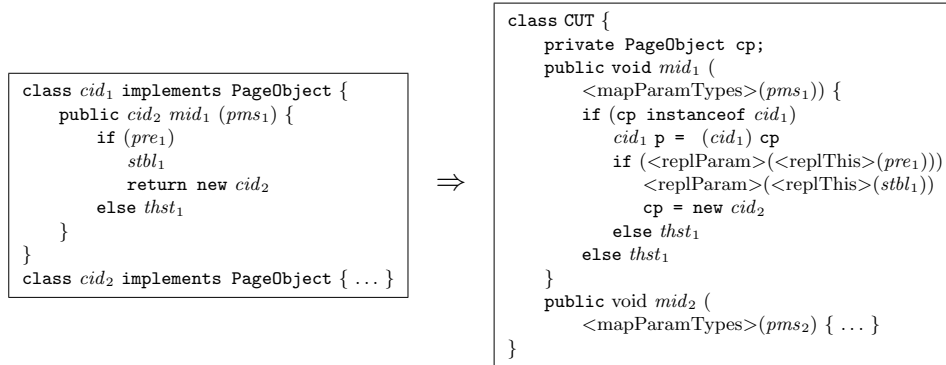
```
class cid₁ implements PageObject {
    public cid₂ mid₁ (pms₁) {
        if (pre₁)
            stbl₁
            return new cid₂
        else thst₁
    }
}
class cid₂ implements PageObject { ... }
```

$\Rightarrow$

```
class CUT {
    private PageObject cp;
    public void mid₁ (
        <mapParamTypes>(pms₁)) {
        if (cp instanceof cid₁)
            cid₁ p = (cid₁) cp
            if (<replParam>(<replThis>(pre₁)))
                <replParam>(<replThis>(stbl₁))
                cp = new cid₂
            else thst₁
        else thst₁
    }
    public void mid₂ (
        <mapParamTypes>(pms₂) { ... }
}
```

Fig. 2: Automated program transformation that generates class CUT from the POs

assigned to local variable p. This variable must replace any occurrence of this in the body of the original method, including its precondition pre_1. This is performed by function <replThis>. The instruction that returns a new PO in the original code is transformed into a statement that assigns such new PO to the class field cp (*current page*) of CUT.

To facilitate the job of the test generator, the original parameter types (e.g., Id:int) are mapped to a type with smaller range (e.g., $Id \in [1 : 6]$) by function <mapParamTypes>. Such a smaller range can be determined by static analysis, in simple cases as those in Figure 1, or can be specified by the tester. As a consequence, any occurrence of the original parameter identifiers must be replaced with an accessor to the parameter value (e.g., x becomes x.value). This is performed by function <replParam>. Figure 3 shows the result of the transformation when it is applied to the PO in Figure 1.

```
1  public class CUT {
2      private PageObject currentPage;
3      public void selectProduct(Id id, Category category) {
4          if (this.currentPage instanceof ProductsPage) {
5              ProductsPage page = (ProductsPage) this.currentPage;
6              if(page.getActiveCategory() == category.value){
7                  page.driver.findElement(By.id("product-" + id.value + "-" + category.value)).click();
8                  this.currentPage = new ProductDetailPage(page.driver);
9              } else { throw new IllegalArgumentException("Invalid parameter values"); }
10         } else { throw new IllegalArgumentException("You are not in the right page"); }
11     }
12 }
```

Fig. 3: Excerpt of CUT generated from the POs of the Shopping Cart example

We apply search based test case generation as instantiated for object oriented systems [9] in order to find the method sequences and parameter values that cover

the last statements of the transformed method bodies, which correspond to the statements returning a new PO in the original methods (i.e., `this.currentPage = new ProductDetailPage(page.driver)` for `selectProduct`). In fact, coverage of all the statements that return the next PO in the navigation is equivalent to covering all the edges in the navigation graph, i.e., to transition coverage. In particular, we use a Genetic Algorithm (GA) and the evolved chromosomes are test cases, i.e., sequences of method calls. The fitness function is the sum of the branch distances of the yet uncovered branches [9]. On the other hand, the standard genetic operators for object oriented test generation do not work properly in our case, because they do not take the structure of the navigation graph into account. Hence, we have defined new crossover and mutation operators, described in the next section. The initial population is obtained by performing multiple random walks on the navigation graph.

4.2   Genetic Operators

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

Crossover: We have defined a crossover operator that works at test case level, in addition to the usual test suite crossover operator [9]. Our new crossover operator is shown in Figure 4a, where the notation $P_i \rightarrow P_j$ above the method name $m_k()$ indicates that method $m_k()$ has PO P_i as starting node and PO P_j as target node. Crossover is straightforward to apply if the cut point selected on the two chromosomes is between method calls that refer to the same PO (in Figure 4a, the cut point between $m_1()$ and $m_2()$ in both chromosomes refer to the same PO, P_1, which is the target of $m_1()$ and the source of $m_2()$). When this does not happen, the two different POs are connected by performing a random walk in the hammock subgraph between them. To ensure reachability during the random walk, head and tail of the new chromosome are possibly shortened, until reachability holds between the two POs.

Mutation: We have maintained the test suite mutation operator [9], but we have modified the delete and insert method call operators, which work at the test case level. An example of how they manipulate the chromosome is provided in figure 4b. The *change* method call operator is applicable only if the alternative method has the same source and target POs as the original method call.

The *delete operator* randomly selects a starting method from the test case and, given the target PO of the selected method (in Figure 4b, method $m_2()$ and target node P_2), it removes all the following method calls until it finds one with a source node that is equal to the target node of the starting method. If it does not find it, it deletes all the methods from the selected point until the end of the chromosome (as in Figure 4b). This operation cannot remove all the statements (at least one, the first method call, is always left), to avoid the generation of an empty test case.

The *insert operator* always starts at the end of the test case (in Figure 4b, method $m_2()$, which has become the last method call after application of the

delete operator) and it selects a method corresponding to a yet uncovered branch (e.g., $m_6()$). Then it performs a random walk on the hammock subgraph between the target node and the source node of the two selected methods (i.e., the hammock subgraph between P_2 and P_5). The path obtained in such random walk is appended to the chromosome (in Figure 4b, methods $m_8()$, $m_4()$, $m_7()$, plus the target method $m_6()$). If the source node of the uncovered method is unreachable from the end of the chromosome, the insert operator fails and does not change the chromosome.

Insert and delete operations balance each other, by extending and shrinking the chromosomes, hence providing a mechanism for bloat control (*bloat* occurs when negligible improvements in the fitness value are obtained by extremely large solutions).
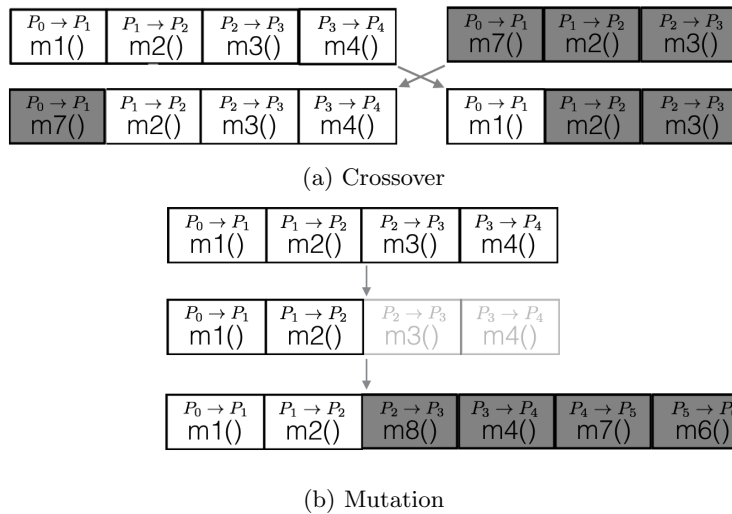
$P_0 \to P_1$	$P_1 \to P_2$	$P_2 \to P_3$	$P_3 \to P_4$		$P_0 \to P_1$	$P_1 \to P_2$	$P_2 \to P_3$
m1()	m2()	m3()	m4()		m7()	m2()	m3()

$P_0 \to P_1$	$P_1 \to P_2$	$P_2 \to P_3$	$P_3 \to P_4$		$P_0 \to P_1$	$P_1 \to P_2$	$P_2 \to P_3$
m7()	m2()	m3()	m4()		m1()	m2()	m3()

(a) Crossover

$P_0 \to P_1$	$P_1 \to P_2$	$P_2 \to P_3$	$P_3 \to P_4$
m1()	m2()	m3()	m4()

$P_0 \to P_1$	$P_1 \to P_2$	$P_2 \to P_3$	$P_3 \to P_4$
m1()	m2()	m3()	m4()

$P_0 \to P_1$	$P_1 \to P_2$	$P_2 \to P_3$	$P_3 \to P_4$	$P_4 \to P_5$	$P_5 \to P_6$
m1()	m2()	m8()	m4()	m7()	m6()

(b) Mutation

Fig. 4: Crossover and mutation (with *delete* followed by *insert*)

5 Empirical Validation

The *goal* of the case study is to assess pros and cons of the proposed approach. The baseline for comparison is the navigation graph produced by a state of the art crawler, Crawljax [10], and the test cases derived from such graph. We have formulated the following research questions:

RQ1 (Cost): *What is the size of the page objects to be written manually and what is the size and complexity of the Page Object method pre-conditions, required by our approach?*

To analyze the manual cost that a tester incurs when using our approach, we measure the lines of code (LOC[4]) of all POs needed to model the subject

---

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

application. In particular, we are interested in the manual cost for writing the preconditions, since they represent a requirement specific of our approach. We measure the total number of preconditions, the total number of logical operators in such preconditions and the lines of code of methods used exclusively by preconditions.

RQ2 (Navigation graph): *How does the navigation graph specified through POs differ from the navigation graph obtained through crawling?*

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

RQ3 (Test suite features): *What is the size of the test suite generated by* SUBWEB *as compared to that derived from the crawled navigation graph and what is the proportion of divergent test cases?*

Test case derivation from the navigation graph produced by *Crawljax* is supported by the tool *Atusa* [1], which is unfortunately unavailable. Hence, we have reimplemented the test derivation algorithm of *Atusa* by following its description in the reference paper [1]. We call our reimplementation *Ext-Crawljax*. We are interested in comparing the size of the test suites produced by SUBWEB vs *Ext-Crawljax*. A smaller size is preferable because it makes manual oracle creation or validation easier for testers. Moreover, we measure the proportion of divergent test cases (i.e., those that upon execution do not cover the test path for which they were generated). In fact, the occurrence of divergences is detrimental to the actually achieved coverage, with respect to the theoretical coverage guaranteed by the test case derivation algorithm.

RQ4 (Coverage): *What is the level of coverage reached by the test cases generated by* SUBWEB *in comparison with the coverage reached by the test cases derived from the crawled navigation graph?*

Regarding coverage, which represents the core objective of test generation, we consider the transition coverage adequacy criterion, measured in the navigation graph specified by testers through POs. This required to manually map states and transitions in the crawled navigation graph to states and transitions in the PO navigation graph.

5.1   Tool

We have implemented SUBWEB on top of *EvoSuite* [9]. In particular, we have enabled the *Whole Test Suite* strategy, because we have multiple targets to satisfy. We have modified *EvoSuite* in order to take the navigation graph into account, both when generating the initial random population of individuals and in the genetic operators, which must generate method sequences compliant with the navigation graph.

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

5.2   Case study

AddressBook[5] is a web-based address and phone book, contact manager and organizer. It is written in *PHP* and it uses *JavaScript* for handling and modifying *HTML* elements at runtime; moreover it is backed by a *MySQL* database. The size of the application, shown in Table 1a, is non trivial. Moreover, this application has been used as a case study in previous works [11].

We have removed a few features from the application, regarding uploads and downloads of files (photos and text files for instance), as well as address locations in a map, since they increase the navigation/testing time while being straightforward to test.

5.3   Experimental Procedure

For the sake of fairness, we granted both tools, SUBWEB and Crawljax, an overall execution budget of 2 hours and we ran both tools on the same subject 10 times, because both tools have non deterministic behaviour. In SUBWEB we have disabled the minimization step of *EvoSuite*, because it requires multiple, costly test case executions on the browser, which makes it too inefficient for our purposes. In Crawljax, we use the default configuration with the default parameter values. We only provide Crawljax with custom values for those form inputs in the application that require very specific values.

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

5.4   Results

The data in Table 1a show that the 13 POs written manually account for 764 LOC in total. This is a small fraction of the overall application size (around 2%). Preconditions, that are required exclusively by our approach, represent an even

---

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

smaller portion of the application size: precondition method LOC account for 0.2% of the application size, while the 16 preconditions use on average 3 logical operator each. Moreover, the first author wrote the 13 POs in, approximately, one day; however this metric clearly depends on many factors, the main one is the level of confidence the developer has with the Page Object pattern.

App	PHP LOC	30223
	JavaScript LOC	1288
POs	LOC	764
	Total number	13
	Navig. methods	73
Preconds	Method LOC	75
	Total number	16
	Logic operators	54

(a) RQ1

PO graph	States	12
	Transitions	73
Crawled graph	States	329
	Transitions	927
	Missing states	0
	Missing trans	5
	Split state ratio	27
	Split trans ratio	13

(b) RQ2

Subweb	Test cases	54
Ext-Crawljax	Test cases	598
	Divergent test cases	104 (17%)

(c) RQ3

Table 1: Size of application, POs and PO preconditions (a); size of PO vs crawled graph, with missing/split states/transitions (b); number of test cases and divergent test cases (c)

> **RQ1**: *Based on the size data collected on our case study, the manual cost for writing POs and PO preconditions seems relatively low.*

As shown in Table 1b, the crawled navigation graph is huge if compared to the PO navigation graph (approximately $\times 27$ states; $\times 13$ transitions). While it does not miss any state, despite its size it misses on average 5 transitions, which are specified by testers, but are not covered during some executions of crawling (5 is the average computed over 10 runs of Crawljax). No single case of state/transition merge was observed, while, as expected from the larger graph size, several states and transitions are split in the crawled graph.

> **RQ2**: *The crawled graph deviates from the ideal, manually specified, PO graph to a major extent, because of its larger size, missing transitions and split states/transitions.*

Table 1c shows that Subweb generates much smaller test suites than Ext-Crawljax. This is a consequence of the different navigation graph size. Moreover, while Subweb generates non divergent test cases by construction, the crawling based approach generates as many as 17% divergent test cases.

> **RQ3**: *The test suites produced by Subweb are approximately 11 times smaller than the test suites produced by Ext-Crawljax. The latter include a relatively large proportion of divergent test cases.*

Figure 5 shows the box plots of the transition coverage reached by SUBWEB and Ext-Crawljax. The mean coverage of SUBWEB is on average 13pp (percentage points) above the mean coverage of Ext-Crawljax and such a difference is statistically significant according to the Mann-Whitney U test (at 5% significance level), that we applied since we didn't have a priori knowledge about the distribution of the data.

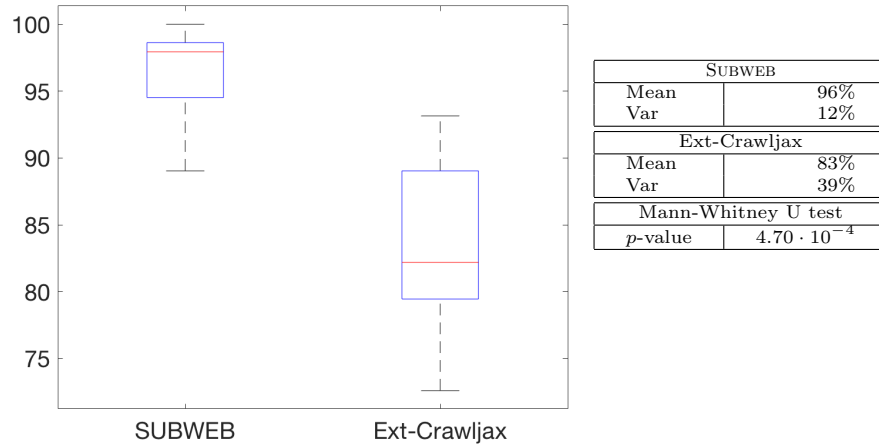| SUBWEB	
Mean	96%
Var	12%
Ext-Crawljax	
Mean	83%
Var	39%
Mann-Whitney U test	
p-value	$4.70 \cdot 10^{-4}$

Fig. 5: Transition coverage (percentage) reached by SUBWEB and Ext-Crawljax in 10 runs; the two distributions differ in a statistically significant way according to the Mann-Whitney U test.

> **RQ4**: *The test cases generated by* SUBWEB *achieve higher transition coverage than those generated by Ext-Crawljax.*

5.5   Threats to Validity

Threats to the *internal validity* might come from how the empirical study was carried out. Each test case was run starting from an empty database, under the assumption that the tester is interested in the behaviour of the application when no record has been persisted yet. If, on the contrary, a non empty database is created at each test case start up, the traversal of paths for which populating the database is a prerequisite becomes easier for both approaches.

Moreover, we didn't use a case study with existing POs and measured the effort needed to modify them in order to enable our technique; indeed it is difficult to find open source projects with existing selenium tests using the PO design pattern.

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

6   Conclusions and Future Work

We have presented Subweb, a web testing tool for the joint generation of test inputs and feasible navigation paths. Although Subweb requires a manual step for POs writing, whereas a crawling-based approach is completely automatic, the effort of such manual step is quite limited while, on the other hand, the achieved advantages are major ones: the navigation graph is much smaller; correspondingly, the test suites derived from the navigation graph have substantially smaller size; by construction, test cases are never divergent, while this is not the case of crawling-based test cases; finally, the transition coverage reached by Subweb is on average higher (96% vs 83%).

In our future work, we will investigate techniques to support the automatic generation of assertions starting from the generated test suite. Moreover, we plan to evaluate Subweb on other web applications, in addition to *AddressBook*.

References

1. Mesbah, A., van Deursen, A.: Invariant-based automatic testing of ajax user interfaces. In: Proceedings of the 31st International Conference on Software Engineering. ICSE '09, Washington, DC, USA, IEEE Computer Society (2009) 210–220
2. Leotta, M., Clerissi, D., Ricca, F., Tonella, P.: Approaches and tools for automated end-to-end web testing. Advances in Computers **101** (2016) 193–237
3. Artzi, S., Dolby, J., Jensen, S.H., Møller, A., Tip, F.: A framework for automated testing of javascript web applications. In: Proceedings of the 33rd International Conference on Software Engineering. ICSE '11, New York, NY, USA, ACM (2011) 571–580
4. Marchetto, A., Tonella, P., Ricca, F.: State-based testing of ajax web applications. In: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation. ICST '08, Washington, DC, USA, IEEE Computer Society (2008) 121–130
5. Tonella, P., Ricca, F., Marchetto, A.: Recent advances in web testing. Advances in Computers **93** (2014) 1–51
6. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: Proceedings of the 29th International Conference on Software Engineering. ICSE '07, Washington, DC, USA, IEEE Computer Society (2007) 75–84
7. Yu, B., Ma, L., Zhang, C.: Incremental web application testing using page object. In: Proceedings of the 2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb). HOTWEB '15, Washington, DC, USA, IEEE Computer Society (2015) 1–6
8. van Deursen, A.: Testing web applications with state objects. Commun. ACM **58**(8) (July 2015) 36–43
9. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering **39**(2) (feb. 2013) 276 –291
10. Mesbah, A., van Deursen, A., Lenselink, S.: Crawling ajax-based web applications through dynamic analysis of user interface state changes. ACM Transactions on the Web (TWEB) **6**(1) (2012) 3:1–3:30
11. Leotta, M., Stocco, A., Ricca, F., Tonella, P.: Robula+: an algorithm for generating robust xpath locators for web testing. Journal of Software: Evolution and Process **28**(3) (2016) 177–204