Dependency-Aware Web Test Generation

Matteo Biagiola*, Andrea Stocco†, Filippo Ricca ‡, Paolo Tonella†

*Fondazione Bruno Kessler, Trento, Italy, biagiola@fbk.eu
†Software Institute - USI, Lugano, Switzerland, {andrea.stocco, paolo.tonella}@usi.ch
‡Università degli Studi di Genova, Genoa, Italy, filippo.ricca@unige.it

***Abstract*—Web crawlers can perform long running in-depth explorations of a web application, achieving high coverage of the navigational structure. However, a crawling trace cannot be easily turned into a minimal test suite that achieves the same coverage. In fact, when the crawling trace is segmented into test cases, two problems arise: (1) test cases are dependent on each other, therefore they may raise errors when executed in isolation, and (2) test cases are redundant, since the same targets are covered multiple times by different test cases. In this paper, we propose DANTE, a novel web test generator that computes the test dependencies associated with the test cases obtained from a crawling session, and uses them to eliminate redundant tests and produce executable test schedules. DANTE can effectively turn a web crawler into a test case generator that produces minimal test suites, composed only of feasible tests that contribute to achieve the final coverage. Experimental results show that DANTE, on average, (1) reduces the error rate of the test cases obtained by crawling traces from 85% to zero, (2) produces minimized test suites that are 84% smaller than the initial ones, and (3) outperforms two competing crawling-based and model-based techniques in terms of coverage and breakage rate.**

***Index Terms*—Web Testing, Crawling-based Test Generation, Dependency Detection, Test Minimization**

I. INTRODUCTION

Crawlers are appealing tools for testers as they can effectively and rapidly explore the state space of a web application. However, deriving executable functional test cases from a sequence of crawled pages and events is not trivial. First, such sequence is typically very long, and it would be inefficient to execute it entirely within a single test case. Therefore, crawlers segment such a sequence into meaningful sub-sequences, according to some criteria [1], [2]. Such sub-sequences better isolate the different functionalities exercised by the crawler, and can be considered as separate test cases. Second, the obtained sub-sequences may have *dependencies* on web app states created by previous sub-sequences [3], [4], [5], which must be resolved. Third, since crawlers are designed to perform continuous, repeated explorations with different randomly generated inputs, the segmented sequences may also be *redundant*, which increases the test suite runtime, without benefiting the overall coverage of the web app functionalities. In our experiments, the segmented crawling trace produced 69 candidate test cases on average, of which 59 were redundant.

Existing works [1], [6], [7], [8] use crawlers to support test generation quite differently. Particularly, navigational model of the web app is used as main driver for the generation of tests. The web pages visited by the crawler upon its exploration are abstracted into DOM *states*, connected by state *transitions* that represent the actions performed by the crawler (e.g., clicking on hyperlinks, or filling in forms). Given some model-based adequacy criterion, such as transition coverage, tools like Atusa [1] derive a set of paths from the navigational model. Such paths represent abstract test cases that can be turned into concrete, executable test cases by supplying proper input values. The main limitation of such model-based test generation techniques is that they have to cope with the *feasibility problem*, i.e., finding proper paths in the navigational model along with associated input values, such that, upon execution of the web application under test, the desired navigation is taken. Determining if a path is feasible is in general undecidable. Infeasibility may be due to dependencies on previous application states that the selected path cannot reproduce or on the impossibility to generate input values satisfying the path constraints [7]. Our experiments show that most (85%) of the paths obtained by segmenting the trace produced by our crawler raise errors if executed in isolation. While not all of them are indeed infeasible, our findings confirm that turning a crawled path into an executable test case cannot be easily achieved by just replaying the crawling path with the same inputs used originally.

In this paper, we propose a novel approach to web test generation that combines *crawling*, *test dependency detection*, and *test minimization*. Our approach, implemented in a tool called DANTE (**D**ependency-**A**ware Crawli**n**g-Based Web **T**est G**e**nerator), takes as input the raw segmented crawling trace produced by a crawler, computes and validates the test dependencies between each sub-sequence, ensuring that any resulting test suite (sub-list of segmented sequences) that respects them will not result in any *test breakage* [9]. Last, DANTE uses test dependencies as a set of constraints for detecting and eliminating redundant tests using a SAT solver.

We evaluated DANTE on five real-world web applications. DANTE generated only breakage-free (i.e., *feasible*) test suites, while the error rate of the test cases obtained by segmenting the initial crawling trace was as high as 85%. Such executable test cases were obtained by discarding as much as 84% redundant test cases from the initial set produced by the crawler, ensuring at the same time that the original code coverage is unchanged. Our results also show that tests minimized by DANTE achieve substantially higher code coverage than two competing techniques (i.e., Crawljax and Atusa).

Our paper makes the following contributions:

- The first dependency-aware crawling-based approach for end-to-end web test generation. Our approach introduces

dependency analysis of the tests obtained through crawling and constraint solving of such dependencies to minimize redundant test cases.
- An implementation of our algorithm in a tool named DANTE [10].
- An empirical evaluation of DANTE on a benchmark of five open-source web applications, including a comparison with two state of the art solutions.

II. BACKGROUND

In this section, we illustrate the functioning of a web crawler, and how it supports the creation of web tests, along with the main limitations on a motivating example.

We use Crawljax [1], [2] as a reference. Crawljax (hereafter referred to as the crawler) has been recognized as the state-of-the-art crawler for web applications by a large empirical study comparing different crawlers [11].

A. Automated Crawling of Web Applications

Crawljax can explore JavaScript web applications through automatic dynamic analysis of UI-state changes in the browser. The crawler scans the DOM tree of each web page, spots candidate elements that are capable of changing the DOM state (i.e., clickables), fires events on those candidate elements, and incrementally creates a graph of the web application in which nodes are the dynamic DOM instances of the web pages, and edges are event-based transitions between them.

Exploration Strategy. The crawler explores the web application according to a graph visit algorithm, such as breadth-first or depth-first visit [12], [13]. The default option in Crawljax is the depth-first visit: on each DOM state, one clickable is selected and the event associated with it is fired. If such event reveals a new DOM state, the visit continues from the newly discovered DOM state. Whenever no more candidate elements are present in a given DOM state, or no new DOM states are discovered in a given crawl path, the crawler backtracks its exploration to the first DOM state containing unexplored candidate elements. The order in which candidate elements are selected within a DOM state can be changed. For instance, FeedEx [6] selects the clickable element to consider based on a linear combination of factors aimed at maximizing the diversity of the exploration.

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

State Abstraction Function. To avoid redundancies, DOM states that are identical or similar to previously encountered DOM states should be discarded. The problem of detecting already visited DOM states is delegated to the state abstraction function, which is a boolean function that decides if a new DOM state is found after an event is fired [14]. The state abstraction function integrated within Crawljax compares the equality of the string representation of the DOM of each web page, which ensures fast comparisons, and therefore, more exploration capabilities. However, other state abstraction functions have been proposed, e.g., comparing the DOM tree by the tree edit distance [6].

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

B. Test Generation supported by Crawling

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

First, after segmentation, the individual test cases may be *dependent* on each other. Test dependencies are due to the web application state being modified by actions performed by the crawler in previously executed test cases [3], [4], [5].

Second, test *redundancy* may appear as a consequence of the length of the navigation performed by the crawler. Long navigations are indeed desirable, because they have more chances to explore the web application in depth. On the other hand, thorough explorations may also include sub-segments that are equivalent according to some chosen adequacy criterion, such as code/model coverage.

Other factors affecting the degree of test redundancy are the crawler's state abstraction function, and the strategy used to select the next DOM element to explore within a DOM state (Section II-A). For what concerns state abstraction, when the state abstraction function is too coarse-grained, many parts of the web application would be unexplored because many DOM states would be considered the same. Conversely, if the state abstraction function is too permissive, many similar DOM

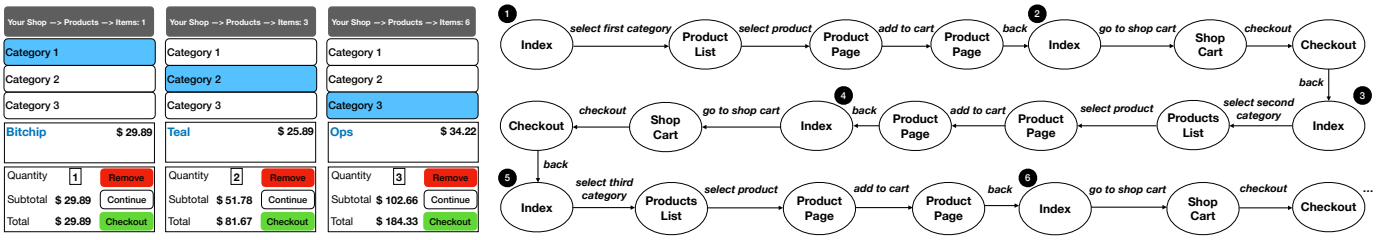

Fig. 1: E-commerce example and generated crawling trace containing dependent and redundant sequences.

states would be considered different, leading to many redundant test cases. For what concerns DOM element selection, when the crawler considers candidate elements *multiple times*, it can explore the web application more deeply, at the cost of potentially increasing the redundancy of the tests.

Model-based Test Generation and its Limitations. The output of Crawljax is the navigational model of the web application, in the form of a state-flow graph of the abstracted DOM states and the event-based interactions between them [1]. Such a model can be used for generating test cases as well, and different solutions have been proposed.

The approach implemented by Atusa [2] extracts test cases from the model using a *k-shortest path* algorithm, a generalization of the shortest path problem in which several paths in increasing order of length are sought for path selection. Specifically, Atusa collects all sinks in the crawled graph, and computes the shortest path from the index page to each of them. Crawl paths are completed by randomly generated inputs, or by reusing the same inputs used during the exploration. A drawback of this strategy is the high chance of producing infeasible test cases. Infeasibility may be due to dependencies on previous application states that the selected path cannot reproduce or on the impossibility to generate input values satisfying the path constraints [7].

C. Motivating Example

Figure 1 (left) shows a sample e-commerce web application. A typical navigation consists in the user selecting one product category, and the desired product in the `Product List` page. Then, in the `Shop Cart` page, the user can indicate the desired number of items, remove unwanted items, or finalize the purchase. The top bar shows the navigation performed by the user on the application, and the number of products currently present in the shopping cart. For instance, in Figure 1 (left), the user selected one Bitchip product from the first category, two Teal products from the second category, and three Ops products from the third category.

Crawling. Figure 1 (right) shows a possible exploration performed by the crawler on our e-commerce web application that results in a long sequence of visited DOM states. For simplicity, suppose the crawler adopts the default depth-first crawling exploration strategy (i.e., clicking the same web elements only once) and a state abstraction function based on DOM string equality.

In the represented sequence, the crawler selects the first category (Category 1) and the corresponding product. Then, it adds the product to the cart, and navigates back to the index page. Then, the crawler accesses the shopping cart and performs the checkout, before returning to the index page. The same crawling path is repeated for the other two categories (Category 2 and Category 3); the corresponding products are also checked out.

Segmentation. Suppose that in our e-commerce example, the crawler segments the crawling trace whenever it reaches the first visited page (e.g., *Index*). In this case, six (6) subsequences are generated, as shown in Figure 1 (the first DOM state of each new sub-sequence is marked by an incremented number). For instance, the first sub-sequence ❶ ends after having added the first product to the shopping cart, whereas the second sub-sequence ❷ ends after having checked out the first product. The other sub-sequences are segmented similarly.

Dependent Tests. After segmentation, the resulting test cases are dependent on the application state being modified by the actions the crawler performed in previously executed tests. For instance, sub-sequence ❷ (i.e., checking out the first product) depends on sub-sequence ❶ (i.e., selecting the first product in the shopping cart). Similarly, sub-sequence ❹ depends on sub-sequence ❸, and sub-sequence ❻ depends on sub-sequence ❺.

If we execute each of those tests in *isolation*, the first test ❶ executes correctly, whereas the second one ❷ breaks. This is due to the initial application state, which does not contain the shopping cart item that the second test needs for checkout, since that item is created by the first test.

On the contrary, if tests obtained after segmentation are executed in the same order in which they were crawled, no breakage occurs (i.e., by running in sequence tests ❶–❻). On the contrary, if we want to generate independent test cases, then test dependencies should be known and properly addressed [4], [5], [15], [16], [17], [18], [19].

Redundant Tests. The test cases shown in Figure 1 are also redundant considering the coverage of functionalities. The first two sub-sequences ❶–❷ cover a scenario in which a product is selected and checked out. Sub-sequences ❸–❹ and ❺–❻ repeat the same scenario, with different data. If such products are retrieved by the server-side of the web application, the overall client-side code coverage would not change when executing these last two tests. Therefore such tests could be removed without affecting the adequacy of the test suite (considering client-side code coverage as adequacy criterion).

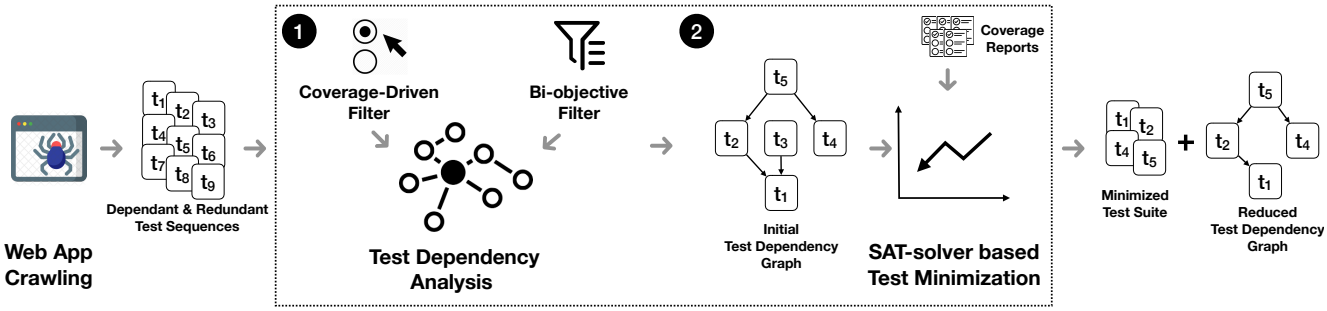

Fig. 2: High-level overview of our approach for dependency-aware web test generation and minimization

III. Approach

Figure 2 illustrates the overall approach, which takes as input a segmented crawling sequence retrieved by a crawler when executed on a given web application (Section II). Test dependency analysis is used to retrieve and validate the dependencies between each sub-sequence (*Test Dependency Analysis*), which are subsequently used as a set of constraints within a SAT solver to eliminate redundant test cases (*SAT solver-based Test Minimization*).

Test dependency analysis is, however, a computationally expensive task, because the total number of possible dependencies is quadratic in the number of test cases. Automatically identifying the entire subset of true dependencies requires exponential analysis time [20], [15], [21]. In fact, in the absence of any preliminary filtering, each test case in the original sequence must be assumed as possibly dependent on all its predecessors, resulting in $n(n-1)/2$ (with n being the number of test cases) candidate dependencies to be validated by dependency analysis, of which only a small subset may actually represent true dependencies.

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

The first filtering heuristic pre-selects only the tests that contribute to web application coverage (*coverage-driven filter*), while discarding the unnecessary ones, and making sure that dependents of the selected tests are included. The second filtering heuristic is based on bi-objective optimization (*bi-objective filter*), where the two objectives being minimized are: (1) the number of dependencies kept for successive dependency validation, and (2) the estimated cost of recovery of the incorrectly filtered dependencies.

In the second phase, test suite minimization aims at eliminating all redundant test cases, i.e., tests that do not contribute to coverage and are not needed due to some dependency.

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

A. Test Dependency Analysis

For dependency retrieval and validation we use an existing tool named TEDD [21], which automatically detects the occurrence of dependencies among web tests. From a given test suite, TEDD retrieves an initial set of dependencies to be validated, which may include both spurious dependencies to be removed as well as missing dependencies to be recovered. The output of TEDD is a test dependency graph such that all test schedules that respect its dependencies execute correctly.

TEDD was proposed for validating dependencies in human-written tests, leveraging filtering techniques based on natural language processing of the test case identifiers [21]. In this paper, we target automatically generated test suites, which do not include meaningful identifiers. In our preliminary experiments, TEDD does not terminate within a timeout of 48 hours (2 days), when applied to the complete (quadratic), unfiltered test dependency graph. Hence, we designed two novel filtering heuristics, aiming at making TEDD applicable to automatically generated test cases, which we describe next.

1) Coverage-Driven Filter: The first proposed filter aims to reduce the number of initial tests, thus making the initial test dependency graph smaller. Specifically, the filter retains only the tests that contribute to the coverage of the web application, along with their dependent tests. The filter is independent from the coverage criterion (e.g., any model/transition or code/branch coverage can be adopted). We hereafter refer to *element* as the unit of coverage specific to the selected coverage criterion (e.g., a transition for transition coverage, or a branch for branch coverage).

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

Fixing Missing Test Dependencies. Algorithm 1 shows our automated procedure for fixing missing dependencies. The algorithm takes as input the selected test suite T_s and the original test suite T_o as generated by the crawler.

Algorithm 1: Test suite fixing algorithm

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

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

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

The algorithm computes the preconditions of *brokenTest* (line 6) by considering all tests in the original test suite T_o that are placed before it, and that are *not* yet included in T_s. The loop (lines 9–17) adds one precondition at a time (initially, *windowLength* = 1), and checks if the test suite with the added precondition executes correctly (line 11). If so, the algorithm terminates (*newBrokenTest* = *null*). Otherwise, the previously added preconditions are removed from T_s (line 16) and the loop (lines 9–17) continues. The loop is interrupted also when *newBrokenTest* follows (hence, it has to replace) *brokenTest* (lines 14–15). The ORDER function computes the index of the test case given as input in the original test suite T_o. Such index defines an order relation between test cases that corresponds to the execution order given by crawler segmentation (Section II-C). If *newBrokenTest* is equal to *brokenTest* (lines 18–19), the size of the window is increased.

Let us consider as example a test suite $T = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9\}$. Suppose that the coverage-driven filter selects t_1 and t_5, hence $T_s = \{t_1, t_5\}$, and that t_5 depends on t_2 and on t_4 (indicated as $t_5 \rightarrow t_2$ and $t_5 \rightarrow t_4$). Algorithm 1 tries to execute T_s but t_5 breaks. The preconditions of t_5 are $\{t_2, t_3, t_4\}$. The algorithm tries to add them one at a time, but let us suppose that each one in isolation is not sufficient to fix t_5. Then, at the end of the first iteration, the window size is increased to 2. The algorithm adds $\{t_2, t_3\}$ first and then $\{t_3, t_4\}$ but none of the two sets of tests fixes t_5, therefore they are both removed from T_s. Finally, the window length is increased to 3, and the tests $\{t_2, t_3, t_4\}$ are added. The broken test t_5 passes, and $T_s = \{t_1, t_2, t_3, t_4, t_5\}$ is the fixed filtered test suite. It can be noticed that T_s contains one spurious dependency, i.e., $t_5 \rightarrow t_3$, to be removed by TEDD during dependency analysis. Such a spurious dependency is a consequence of the adopted heuristics, which adds all test cases inside the current window, rather than taking all the possible subsets of the current window (the latter would incur an exponential computational cost).

2) Bi-objective Filter: The second proposed filter aims at reducing the number of dependencies from the complete (quadratic) graph. The general idea is to execute TEDD on a filtered graph that contains as few dependencies as possible, while at the same time minimizing the estimated (worst case) recovery cost that TEDD may incur when identifying and re-introducing missing dependencies. Let $T = \{t_1, \ldots, t_n\}$ be the set of all tests, and let $D = \{d_{11}, \ldots, d_{nn}\}$ be the set of all dependencies associated with the original crawl order. Each dependency d_{ij} is defined as:

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

Let $S = \{s_{11}, \ldots, s_{nn}\}$ be a filtering matrix over D where s_{ij} is 0 if the dependency d_{ij} is filtered; and 1 otherwise ($s_{ij} = 0$ for all $j \geq i$). For each test t_i with $2 \leq i \leq n$, let us consider all outgoing dependencies that are filtered in S (i.e. $s_{ij} = 0$, while $d_{ij} = 1$). Let $D_d(S_i)$ be the set of filtered dependencies of node i which are *direct*, i.e. there is no other path in S that could connect such pair of tests. Let $I_d(S_i)$ be the set of filtered dependencies of node i which are *indirect*, i.e., there exists at least one path in S between such pair of tests. By construction, these two sets are disjoint ($D_d(S_i) \cap I_d(S_i) = \emptyset$). Correspondingly, we define the following two objective functions:

$$\textit{minimize} \quad \textit{deps}(S) = \sum_{s_{ij} \in S} s_{ij} \tag{1}$$

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

Equation 1 counts the number of unfiltered dependencies in S; such sum has to be minimized. Equation 2 estimates the cost of recovery of each dependency that is filtered in S. In the worst case, all filtered direct ($D_d(S_i)$) and indirect ($I_d(S_i)$) dependencies are true dependencies, whose recovery cost is quadratic [21]. The second contribution to the cost of Equation 2 is halved because each indirect filtered dependency may already be included in the graph through other dependencies.

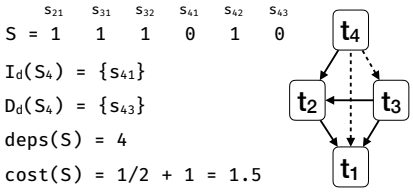

Fig. 3: Bi-objective filtering example. s_{ij} with $i = j$ are not represented since they are 0 by definition.

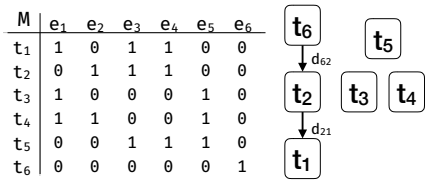

Fig. 4: Minimization example

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

Figure 3 shows an example of how the two objective functions are computed. S is the considered dependency filter that is illustrated by the dependency graph on the right hand side of the figure. Dashed arrows represent filtered dependencies ($s_{ij} = 0$). There are six dependencies in S, and only two are filtered, hence $deps(S) = 4$. The first one s_{41}, which corresponds to $t_4 \rightarrow t_1$, is a filtered indirect dependency, because there is a path between t_4 and t_1 that passes through t_2. If $t_4 \rightarrow t_1$ were true, it would have to be recovered only if $t_4 \rightarrow t_2$ or $t_2 \rightarrow t_1$ are deemed as invalid dependencies during validation. This is why the cost of recovering an indirect dependency has a halved weight than the cost of recovering a direct dependency. The second one s_{43} is a filtered direct dependency which has a recovery cost of one.

B. SAT solver-based Test Minimization

Our approach adopts a SAT solver [22] to find optimal solutions to the test suite minimization problem. We encode the test dependencies as a set of pseudo-boolean constraints that are translated to a SAT instance. Then, the SAT instance is solved using a SAT solver.

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

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

The objective of minimization is to find a subset of tests $X \subseteq T$ with minimum cost such that (1) all elements in E are covered, and (2) the validated dependencies $D = \{d_{ij}\}$ between tests are respected. Formally:

$$\textit{minimize} \quad \sum_{i=1}^{n} c_i t_i \tag{3}$$

subject to:

$$\sum_{i=1}^{n} m_{ik} t_i \geq 1, \qquad 1 \leq k \leq l \tag{4}$$

$$\forall d_{ij} \in D,\ d_{ij} = 1 \wedge t_i = 1 \implies t_j = 1 \tag{5}$$

Equation 3 is the objective function. It is a linear combination of selected tests along with the related execution cost coefficients. The goal is to find a test suite having the smallest execution cost. Equation 4 represents the coverage constraints, one for each element e_k to be covered. Each coverage constraint specifies that at least one test covering each element e_k must be included in the final test suite X. Equation 5 represents the dependency constraints, one for each validated dependency $d_{ij} = 1$. The dependency constraint states that if a dependee test t_i is included in the final test suite X, then its dependent t_j must be included as well.

For example, let us consider $T = \{t_1, t_2, t_3, t_4, t_5, t_6\}$ and $C = \{3.0, 3.0, 2.5, 3.5, 4.0, 1.0\}$. Figure 4 shows the coverage matrix M on the left hand side and the test dependency graph *TDG* on the right hand side. Below, we list both the coverage constraints, on the left hand side, and the dependency constraints, on the right hand side:

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

The first coverage constraint specifies that at least one of the tests $\{t_1, t_3, t_4\}$ must be included in the solution because such tests cover the element e_1. The first dependency constraint is related to the dependency d_{21} in the dependency graph shown in Figure 4. The dependency states that $t_2 \rightarrow t_1$, hence if t_2 is included in the solution ($t_2 = 1$), then the tests t_2 depends on, i.e. t_1, must be also included as well ($t_1 = 1$).

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

C. Implementation

We implemented our approach in a tool called DANTE (**D**ependency-**A**ware, Crawli**n**g-Based Web **T**est G**e**nerator) [10]. The tool is written in Java, and generates Selenium WebDriver Java web test suites. We used Crawljax 4.1 [23] for generating crawling-based test suites as input to our tool. DANTE integrates *TEDD* [21] to compute the test dependencies, the Java bindings of the Z3 SAT solver [24] (4.8.4) to compute the minimization, and *NSGAII* [25] for the bi-objective optimization. The output of DANTE is a minimized test suite and a list of validated dependencies.

IV. EMPIRICAL EVALUATION

A. Research Questions

To assess the practical effects of dependency analysis and test suite minimization, we consider the following research questions:

RQ_1 (breakage rate). *What is the breakage rate of the segments generated by Crawljax?*
RQ_2 (minimization). *What is the test suite minimization rate achieved by* DANTE*?*
RQ_3 (performance). *What is the runtime of* DANTE*? What is the contribution of coverage-driven filtering and bi-objective filtering in making test dependency validation efficient?*
RQ_4 (coverage). *How does* DANTE *compare to Crawljax and Atusa in terms of code coverage and breakage rate?*

RQ_1 aims at quantifying the breakage rate of the traces retrieved by crawling a web application. RQ_2 aims at assessing the improvement achieved over the initial breakage rate, as well as over the initial test suite size. RQ_3 concerns evaluating the tool's performance, and the effectiveness of the two proposed filters to reduce the test dependency graph. The final research question RQ_4 compares the test suites by DANTE with those obtained by two crawling-based techniques, namely Crawljax and Atusa (Section II-B), in terms of code coverage.

B. Subjects

Modern web applications take advantage of the functionalities offered by existing powerful JavaScript frameworks. We selected open source single-page web applications (SPAs) that were used in the context of previous research on web testing, which were available [26]. Particularly, subjects were chosen for their representativeness, recency, size, and popularity.

The considered web apps are quite mature (number of commits ≥ 99), and have been maintained recently (year of last commit ≥ 2016). The size of the systems ($> 1k$ client-side JavaScript LOCs, frameworks excluded) is representative of modern web applications [27].

C. Procedure and Metrics

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

To answer our research questions, we evaluated *two different configurations* of DANTE, by enabling in turn only one of the proposed filtering mechanisms—*coverage-driven filter*, and *bi-objective filter*—followed by test suite minimization. We do not report results for the configuration in which both filters are applied because we found experimentally that the latter becomes useless if the former is applied.

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

We evaluated *performance* (**RQ_3**) by measuring the execution time (in minutes) of each step of DANTE. To assess the impact of the filtering heuristics (i.e., coverage-driven filter *vs* bi-objective filter) on the test dependency validation time, we executed both configurations of DANTE on each subject, and compared the running time to complete the dependency validation in each configuration.

Concerning *coverage* (**RQ_4**), we compared DANTE against Crawljax and Atusa [1], a state-of-the-art model-based web test generator, in terms of client-side coverage and breakage rate (only tests that do not break during execution are considered for coverage). Measuring client-side coverage is meaningful in SPAs since most of the business logic of the application resides on the client. We configured Atusa to use the same inputs used during crawling. Moreover, we chose the value of k for the k-shortest path algorithm applied in Atusa by varying its value in the range [1,1000]. We selected the value of k which generated the highest number of tests in such interval, constrained the test suite runtime to 60 minutes. Such number was estimated by proportion over the test suite runtime when $k = 1$. To measure coverage, we used *cdp4j* 3.0.8 [28], the Java implementation of Chrome DevTools [29], which outputs the *byte coverage* of the executed client-side JavaScript code reached by each test case.

D. Results

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

The minimum breakage rate occurred for *petclinic* (24 tests that break), whereas for the remaining subjects almost all test cases broke (96–98% breakage rates). These empirical results show that tests generated from the output of a crawler

TABLE I: Results for Breakage Rate (RQ_1), Minimization (RQ_2), Performance and Filtering Heuristics (RQ_3).

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

cannot be executed without taking into account the hidden test dependencies due to their shared web application states.

We manually investigated such high breakage rates and found that they are due to application states created by the initial tests, which the next tests rely upon. For example, in *retroboard*, *phoenix* and *dimeshift*, the first tests perform a login operation; the subsequent tests must be executed in an application state (browser and database state) in which the login has already been performed. In *splittypie*, an expense splitting application, the second test creates an event for splitting expenses among participants. After the event is created, the event page view becomes the new application home page (a refresh redirects the browser to the event page), and all tests executed after the second one expect such page as starting page (unless a test case removes that event). *petclinic*'s tests have a lower breakage rate since no login functionality is present. Nevertheless, its tests still exhibit breakages due to shared application states produced by previous tests in other parts of the web application (for instance, a certain pet must be created prior to schedule a visit with a veterinarian).

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

RQ_2 (minimization). Table I (macro-column Minimization) shows the minimization results achieved by both configurations of DANTE on our subjects. For each configuration, the table shows the number of tests removed at each step of the approach (coverage-based/bi-objective filter; test suite minimization), the final test suite size, and the minimization rate achieved with respect to the initial test suites. In the first configuration (Coverage-driven filter & Minimization), the main contribution to minimization is given by the coverage-driven filter (columns 5–6), which removed 84% of the initial test cases on average, with minimization scores greater than 70% across our subjects. The greatest minimization score occurred for *retroboard* (96%), whereas the lowest occurred for *phoenix* (72%). SAT solver-based minimization (columns 7–8) reduced, on average, only by 1% the original test suite size. Therefore, in this configuration, the coverage-driven filter contributes substantially more than the minimization step to the reduction of the size of the initial test suites. Overall, in the first configuration, the final test suites generated by DANTE for our subjects are 84% smaller than the initial ones generated by Crawljax (columns 9–10).

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

The two evaluated configurations of DANTE achieve similar minimization scores on our subject systems (84% *vs* 81%). The difference is explained by the different instances of the problem that the SAT solver must solve, and by the functioning of TEDD, which does not ensure having a minimal test dependency graph. The set of dependency constraints is different among configurations because the two different filtering techniques produce two different initial test dependency graphs (in terms of number of tests, hence in terms of dependencies). Consequently, TEDD produces two slightly different, yet valid, test dependency graphs. Thus, the SAT solver formulation takes into account two different sets of dependency constraints.

RQ$_3$ (performance). Table I (macro column Performance) shows the running time (in minutes) of each step of DANTE for all subject systems and for both configurations. The coverage-driven filter, which includes greedy coverage-driven test selection and test suite fixing (Algorithm 1), takes 30.7 min on average (columns 14–16). Most of such execution time is devoted to test suite fixing, whereas the cost of the greedy coverage-driven test selection is negligible (order of few seconds per subject). The fastest execution of the coverage-driven filter occurs for *retroboard* (25 seconds), while the slowest occurs for *dimeshift* (51 minutes).

On average, dependency validation takes as much as coverage-driven filtering (29 minutes), whereas the cost of the minimization step is negligible (18 seconds). Recall that the dependency validation runtime grows more than linearly with the number of dependencies in the test dependency graph, and such number is reduced due to filtering. For instance, in *phoenix*, 15 (53−38) tests are retained after the coverage-driven filter, whereas in *retroboard* only five (5) are left (100−95). In the former case, dependency validation takes 58 minutes, whereas in the latter case it takes 2 minutes. Overall, the first configuration of DANTE takes on average 60 minutes (1 hour) to compute the final minimized test suite (column 17).

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

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

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

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

TABLE II: Comparison between DANTE, Crawljax and Atusa in terms of client-side byte coverage and breakage rate (RQ$_4$).

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

RQ$_4$ (coverage). Table II compares the best configuration of DANTE (Coverage-driven filter & Minimization) with Crawljax and Atusa in terms of client-side byte coverage and breakage rate. On average, the final test suites generated by DANTE are composed of 10 tests, whereas those generated by Crawljax and Atusa contain 69 and 128 tests, respectively (+590% and +1180%). Concerning the breakage rates, 53% of tests generated by Atusa broke when executed in isolation, whereas *none* of the tests generated by DANTE breaks. This means that a substantial proportion of test cases generated by Atusa has to be discarded because they are *infeasible*, hence those tests do not contribute to increase the coverage of the application under test. Overall, tests generated by DANTE have a coverage increase of 33% with respect to Crawljax, and 18% with respect to Atusa.

V. DISCUSSION

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

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

Filtering Techniques. Both evaluated configurations of DANTE have shown significant effectiveness (**RQ$_2$**). Moreover, both proposed filtering techniques allowed reducing the cost of test dependency validation (**RQ$_3$**), which is known to be in general a computationally expensive step, and our work makes no exception. To this aim, the coverage-driven filter has shown better results than the filter based on bi-objective optimization. The reason behind this is the huge size of the initial test dependency graph (on average, 2,453 dependencies). The coverage-driven filter allowed removing many false dependencies *prior* to the expensive validation phase, which explains the 28× time speed-up over the bi-objective filter.

Limitations. Our approach assumes that tests execute deterministically. DANTE does not include a procedure to automatically fix the flakiness of the test cases generated by the crawler, which is a nontrivial task. For instance, simply adding `wait` statements systematically within the test code may unnecessarily and artificially increase the runtime of the test suite, and it may not work when tests are executed on different browsers or hardware configurations. For such reasons, in our experiments, test flakiness was fixed manually after the crawling step. Moreover, test cases generated by DANTE do not include any explicit functional oracle, such as test case assertions. Hence, only the implicit assertions (web application crashes or runtime errors) can expose faults in the web applications under test, unless the automatically generated tests are augmented with manually written assertions. It would be, however, possible to automatically generate assertions that capture the observed (instead of the intended) behaviour for regression testing.

Threats to Validity. Using a limited number of subject systems in our evaluation poses an *external validity* threat, in terms of generalizability of our results. We selected five single-page web applications used in previous web testing works [26]. Such systems are developed with popular frameworks and pertain to different domains, which should guarantee a certain degree of generalizability, although more subject systems are needed to fully address the generalization threat.

Threats to *internal validity* might come from confounding factors of our experiments. We compared all competing algorithms under identical parameter settings. Our choice of Crawljax as sole baseline for crawling-based test suites might pose another threat, as well as Atusa, which is also based on the navigational model provided by Crawljax. However, Crawljax is a well-known and maintained research tool, and, to our knowledge, no better alternatives have been proposed yet. For **RQ$_4$**, we adopted a tool that computes *byte coverage*, instead of the classical statement or branch coverage. However, byte coverage is a fine-grained and precise coverage metric, which can be turned into more coarse-grained coverage metrics, e.g., line coverage, if needed. With respect to *reproducibility*, the source code of DANTE and all subject systems are available in our replication package [10], making our evaluation and results fully reproducible.

VI. RELATED WORK

Automated web testing techniques have received much attention in recent years [1], [8], [30], [7], [31]. Due to space reasons, we outline only the techniques that focus on end-to-end web testing.

We have already described and evaluated Atusa [1], which uses Crawljax to derive a state-flow graph consisting of DOM states and transitions that model the web application under test, and uses such model to generate test cases with predefined invariants as test oracles.

Other approaches use manually generated page object models to guide the test generation [7], [26], [8]. Specifically, SubWeb [7] uses a search-based approach in which model exploration and input data generation are handled jointly. DIG [26], on the other hand, uses input and path diversity to generate tests. Lastly, InwertGen [8] proposes an incremental two-steps algorithm in which page object creation and test generation using Randoop [32] are intermixed.

Artemis [30] is a framework for automated testing of JavaScript web applications based on feedback-directed random testing [32]. Thummalapenta et al. [31] present a behavioural-driven technique for generating web tests along interesting business-related behaviours of the web app. The behaviours of interest are specified in the form of business rules. Marchetto et al. [33] proposed the use of a combination of dynamic and static analysis to model the web application as a finite state machine, and proposed a coverage criterion based on the notion of semantically interacting events.

Differently, DANTE is the only approach that analyzes automatically generated tests produced by a crawler, with the aim of producing smaller test suites. Unlike mentioned techniques, our approach is completely automated, and does not rely on the navigational model of a web application, or a page object model. Rather, it directly turns the raw output of a crawler into executable test cases by reusing the same inputs used upon crawling, resolving dependencies and eliminating redundancies jointly.

VII. CONCLUSIONS AND FUTURE WORK

Web crawlers have long been adopted to generate test cases for web applications, mostly following a model-based approach. In this paper, we propose a novel approach to web test generation, implemented in a tool called DANTE, that transforms the output of a crawler into executable test cases. DANTE analyses the test sequences produced by a crawler and determines the test dependencies occurring between pairs of test cases, while removing redundant tests by means of a SAT-based minimization step.

Our experimental results show that test suites generated by DANTE are 84% smaller on average than the ones produced by the crawler, and never exhibit test breakages. In our future work, we plan to investigate how the internal factors of the web crawler, as the state abstraction function, segmentation and exploration strategy, impact the redundancy and the dependency between tests generated during crawling.

REFERENCES

[1] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-based web applications through dynamic analysis of user interface state changes," *ACM Transactions on the Web*, vol. 6, no. 1, pp. 3:1–3:30, 2012.

[2] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 35–53, 2012.

[3] A. Gyori, A. Shi, F. Hariri, and D. Marinov, "Reliable testing: Detecting state-polluting tests to prevent test dependency," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 223–233. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771793

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

[5] W. Lam, S. Zhang, and M. D. Ernst, "When tests collide: Evaluating and coping with the impact of test dependence," University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-15-03-01, Mar. 2015.

[6] A. Milani Fard and A. Mesbah, "Feedback-directed exploration of web applications to derive test models," in *Proceedings of the International Symposium on Software Reliability Engineering*, ser. ISSRE '13. IEEE Computer Society, 2013, pp. 278–287. [Online]. Available: http://www.ece.ubc.ca/~amesbah/docs/issre13.pdf

[7] M. Biagiola, F. Ricca, and P. Tonella, "Search based path and input data generation for web application testing," in *International Symposium on Search Based Software Engineering*. Springer, 2017, pp. 18–32.

[8] B. Yu, L. Ma, and C. Zhang, "Incremental web application testing using page object," in *Proceedings of the 2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, ser. HOTWEB '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1109/HotWeb.2015.14

[9] A. Stocco, R. Yandrapally, and A. Mesbah, "Visual web test repair," in *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '18. ACM, 2018.

[10] "Dependency-aware web test generation and minimization." https://github.com/matteobiagiola/ICST20-submission-material-DANTE, 2020.

[11] G. L. Breton, N. Bergeron, and S. Hallé, "A reference framework for the automated exploration of web applications," in *Proceedings of the 19th International Conference on Engineering of Complex Computer Systems*, ser. ICECCS '14, Aug 2014, pp. 81–90.

[12] A. Mesbah, *Advances in Testing JavaScript-based Web Applications*, ser. Advances in Computers. Elsevier, 2015, vol. 97, ch. 5, pp. 201–235. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0065245814000114

[13] P. Tonella, F. Ricca, and A. Marchetto, "Recent advances in web testing," *Advances in Computers*, vol. 93, pp. 1–51, 2014.

[14] R. Yandrapally, A. Stocco, and A. Mesbah, "Near-duplicate detection in web app model inference," in *Proceedings of 42nd International Conference on Software Engineering*, ser. ICSE '20. ACM, 2020, p. 12 pages.

[15] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *Proceedings of the 2018 IEEE Conference on Software Testing, Validation and Verification*, April 2018, pp. 1–11.

[16] J. Bell and G. Kaiser, "Unit test virtualization with VMVM," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. ACM, 2014, pp. 550–561. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568248

[17] G. Rothermel, R. J. Untch, and C. Chu, "Prioritizing test cases for regression testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 929–948, Oct. 2001. [Online]. Available: https://doi.org/10.1109/32.962562

[18] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 211–222. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771784

[19] A. Vahabzadeh, A. Stocco, and A. Mesbah, "Fine-grained test minimization," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. ACM, 2018, pp. 210–221. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180203

[20] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe java test acceleration," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 770–781. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786823

[21] M. Biagiola, A. Stocco, A. Mesbah, F. Ricca, and P. Tonella, "Web test dependency detection," in *Proceedings of 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '19. ACM, 2019, pp. 154–164.

[22] F. Arito, F. Chicano, and E. Alba, "On the application of sat solvers to the test suite minimization problem," in *International Symposium on Search Based Software Engineering*. Springer, 2012, pp. 45–59.

[23] "Crawljax: Crawling JavaScript-based Ajax web applications." https://github.com/crawljax/crawljax, 2019.

[24] "The z3 theorem prover." https://github.com/Z3Prover/z3, 2019.

[25] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[26] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Diversity-based web test generation," in *Proceedings of 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. ACM, 2019, pp. 142–153.

[27] F. S. Ocariza Jr, K. Pattabiraman, and A. Mesbah, "Detecting unknown inconsistencies in web applications," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 566–577.

[28] "Chrome DevTools protocol for java." https://github.com/webfolderio/cdp4j, 2019.

[29] "CSS and JS code coverage." https://developers.google.com/web/updates/2017/04/devtools-release-notes#coverage, 2019.

[30] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip, "A framework for automated testing of javascript web applications," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 571–580. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985871

[31] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra, "Guided test generation for web applications," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 162–171. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486810

[32] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.37

[33] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of Ajax web applications," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ser. ICST '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 121–130. [Online]. Available: http://dx.doi.org/10.1109/ICST.2008.22